

Rich web applications with OpenLaszlo

LASZLO in Action

Norman Klein
Max Carlson
with Glenn MacEwen

SAMPLE CHAPTER





Laszlo in Action

Norman Klein
Max Carlson
with Glenn MacEwen

Chapter 16

Copyright 2008 Manning Publications

brief contents

PART 1 THE BASICS 1

- 1 ■ Turbocharging web technology 3
- 2 ■ The declarative world of LZX 17
- 3 ■ Core LZX language rules 48
- 4 ■ A grand tour of views and user classes 78
- 5 ■ Designing the Laszlo Market 114

PART 2 PROTOTYPING THE LASZLO MARKET 149

- 6 ■ Laying out the Laszlo Market 151
- 7 ■ Introducing Laszlo components 172
- 8 ■ Dynamic behavior of events and delegates 199
- 9 ■ Using Laszlo services 220

PART 3 LASZLO DATASETS 249

- 10 ■ Working with XML datasets 251
- 11 ■ Using dynamic dataset bindings 281
- 12 ■ Scoreboarding the shopping cart 319

PART 4 INTEGRATING DHTML AND FLASH..... 351

- 13 ■ Enhancing the user experience 353
- 14 ■ Branding an application 385
- 15 ■ Integrating DHTML and Flash 404

PART 5 SERVER AND OPTIMIZATION ISSUES 435

- 16 ■ Networked data sources 437
- 17 ■ Managing large datasets 459
- 18 ■ Laszlo system optimization 484

Part 5

Server and optimization issues

The previous parts of this book promoted postponing server-side development by relying on resident datasets for early testing. This approach severs dependencies between the front-end and server-side development groups. Consequently, each group can work independently toward a common API, which serves as the bridge between their efforts. Although some development groups might find it more productive to work jointly in early development, that choice should be an option and not imposed by the working environment.

Our next objective is to make the transition to an integrated development effort as seamless as possible. Chapter 16 provides a methodical approach to update resident datasets into HTTP-supported datasets. Once we have access to large server-supported databases, we'll need to worry about performance optimization issues. The Laszlo Market will be updated to support sessioned data—that is, to maintain shopping cart contents between sessions. Chapter 17 demonstrates methods for managing large datasets in Laszlo to ensure that resources aren't wasted. In particular, the Market will be updated to display large amounts of data using paged datasets. Chapter 18 deals with optimizing an application's startup time with dynamic libraries and redistribution of initialization costs over time.

16

Networked data sources

This chapter covers

- Converting resident to HTTP datasets
- Building data services
- Building a sessioned application
- Maintaining server domains

Be silent as to services you have rendered, but speak of favors you have received.

—Seneca, Roman philosopher

We've arrived at a point where the major Laszlo design issues have been resolved and further development requires access to a networked data source. Up to this point, we've used resident datasets to supply all of our data-related needs. Although networked data could have been introduced earlier, we've purposely postponed it. Decoupling client and back-end development allows each to proceed independently, an approach that has a number of advantages. We enjoyed unlimited freedom to experiment with interface development without support from a server development group. Our client development front end was also isolated from any possible back-end problems, which simplified and sped up development. Finally, although we didn't use this capability, this approach allows extensive unit testing early in development, which can be reused for the server implementation.

Although local datasets initially sped up our development, they are constrained by capacity and nonpermanent persistence. To remove these constraints, it's necessary to work with networked resources. Unfortunately, this transition complicates processing. Since our data resources are no longer compiled into the application, we can't be assured that our data is always available. We must now deal with the fallibility issues of networked data, including server failure and congestion issues that result in error and timeout conditions.

This transition must be as effortless and seamless as possible. In chapters 8 and 12, we used local datasets to establish the XML data structure for the `dsProducts` and `dsCart` datasets. This XML data structure was used as a model for creating our data path's XPath expressions. Now, we must ensure that data returned by the server has an identical XML structure, as any differences will cause these expressions to return incorrect matches. We want to avoid the pain of debugging broken XPath expressions. The best approach is to add unit testing to ensure compliance.

In this chapter, we'll demonstrate how to create supporting classes that allow an easy transition from local to networked data. These classes can be added to our existing application with minimal code changes. While it isn't quite as easy as flipping a switch, this transition is easily applied to most applications with minimal pain and heartache.

16.1 Interfacing to web servers

To interface to a web server, Laszlo requires only an XML-over-HTTP service, which returns an XML document for a resident dataset. XML-over-HTTP is supported by most servers.

The equivalence between resident- and HTTP-server-supplied data makes comparison easy. If an HTTP-supplied dataset matches a resident dataset that was used in early development, then the data path XPath statements in the application remain valid. But before we can start the transition from resident to HTTP datasets, we need to review XML-over-HTTP and dataset operation.

The HTTP standard specifies that all HTTP servers return a response whose content type can be set to `text/xml`. This allows Laszlo to interface to any HTTP web server—ranging from basic HTTP web servers such as Apache, Jetty, or Microsoft's Internet Information Services (IIS) to the various web frameworks such as Struts, Tapestry, Ruby on Rails, and Microsoft .NET—that work with servlet containers to provide enterprise-class services.

There are two approaches to providing XML-over-HTTP web services: they can be either activity or resource oriented. An *activity-oriented* approach views the Internet in terms of available services. A popular example of this approach is Simple Object Access Protocol (SOAP). The *resource-oriented* approach, also known as Representational State Transfer (REST), takes the opposite approach and views the Internet in terms of resources. These REST resources can be manipulated with standard HTTP commands: POST, GET, PUT, and DELETE. We've decided to use the REST approach in this book, since it's an extension of existing best practices and provides the simplest way to interface Laszlo to an HTTP server.

A good example of a REST-based web service is Really Simple Syndication (RSS). RSS newsfeeds are a familiar feature on the Web as they aggregate syndicated web content such as news reports into a list of headlines. These aggregated headlines provide coverage to almost any topic of interest. RSS output is a dialect of XML, which can be retrieved through a standard URL. For example, to access the top stories from Yahoo!, we'd type in the following URL:

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

It will return an XML document whose data composition is defined by the RSS 2.0 specification:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<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>
```

This ensures that the XML document delivered by any RSS 2.0–compliant server has a compliant composition.

We'll start by demonstrating how a dataset can be used to directly retrieve data from a short list of RSS websites and then examine the limits of this approach. Next, we'll see 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 automatically invoke its response handler methods. By the end of this chapter, we'll have a general approach for handling HTTP data that is easily extended to work with any request.

16.1.1 Using datasets with HTTP

Before you can make the transition from static resident to dynamic HTTP datasets, you need a good grasp of dataset properties, particularly those concerned with HTTP. These properties are needed to deal with the dynamic timing involved in loading data from an HTTP server. With that understanding, we'll start with the simplest type of HTTP connection, interfacing to an RSS server.

A dataset can store local data compiled into an application or HTTP data returned by a web server. From Laszlo's viewpoint, data is data; its origin is immaterial.

An application can have any number of datasets. While datasets support every configuration feature in the HTTP standard, we'll cover only their most important features.

A dataset is an unusual Laszlo object since it is descended from multiple parents. It's derived from the `LzNode` object, which means it can be used as a declarative tag, as well as from the `LzDataElement` object (to gain access to its data-access methods). A dataset is flexible enough to supply a simple interface for accessing local data while also supporting all the optional settings defined by the HTTP

standard. A dataset's `src` attribute, which determines the location of the data, 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's operating mode is based on its `src` attribute. If the attribute is omitted or set to a local file, then it's a local dataset; when set to a URL, it interfaces to HTTP-based data. This allows an application's development cycle to easily transition from local test data to networking with an HTTP server.

By default, Laszlo 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 body, since some servers have limits on the size of a GET query string. Tables 16.1 and 16.2 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 Laszlo website at <http://www.Laszlo.org/lps4/docs/reference>.

Table 16.1 Commonly used dataset attributes

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

Table 16.2 Commonly used dataset methods

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

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

To ensure that an application is initially populated with data, the `request` attribute for its datasets should be set to true. This sends a series of initial HTTP requests to populate each dataset when application initialization completes. An HTTP dataset can be manually populated by calling its `doRequest` method.

We'll now use the attributes and methods listed in tables 16.1 and 16.2 to demonstrate how easy it is to implement an RSS newsfeed with Laszlo.

Implementing an RSS newsfeed

Data paths and datasets make accessing RSS information simple. Listing 16.1 sets the `request` attribute for the newsfeed dataset, so it's initialized with data to create a headline listing display. Since the data contained in these feeds is RSS-compliant,

their data composition can be expressed with a single set of data path expressions. We can change the `src` attribute of our dataset to point to any other RSS 2.0-compliant newsfeed with the assurance that our data path expressions still work correctly. In our example, we'll toggle between the CNN and Yahoo! newsfeeds by clicking the Change button. This example could easily be expanded to include additional selections.

Listing 16.1 Accessing RSS newsfeeds

```

<canvas>
  <dataset name="newsfeed" request="true"
    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"
);}
      else
newsfeed.setSrc("http://rss.news.yahoo.com/rss/topstories");
newsfeed.doRequest();
      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>

```

① Generates initial display

② Changes news source

③ Sends request to server

The dataset's `request` attribute ① generates the initial display of the Yahoo! RSS news headlines, as seen in figure 16.1. Clicking the Change button updates the `src` attribute ② to display the URL for CNN's RSS feed. After this update, the `doRequest` method ③ sends the request to the server.

Although a single dataset works adequately in this trivial example to demonstrate the basic dataset features, the shortcoming of this approach becomes apparent when we deal with real-world Internet problems such as HTTP servers timing out or

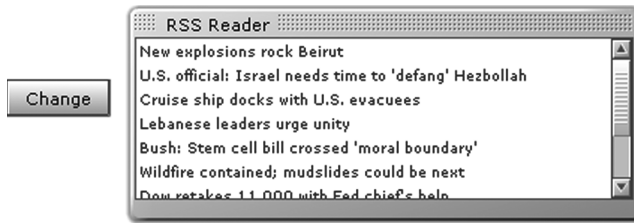


Figure 16.1
Titles from the Yahoo! RSS
newsfeed are displayed
initially. A click of the
Change button switches
the RSS newsfeed to CNN.

being down. In these cases, our RSS reader would lose its current display listing, and only an HTTP server exception code such as HTTP 404 or HTTP 500 would be returned. Providing a seamless viewing experience requires that buffered datasets be used to handle HTTP transfers.

16.1.2 Buffered HTTP 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 pair 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 in a `handleData` method, errors in a `handleErrors` method, and timeout situations in a `handleTimeout` method:

```
<dataset name="dsBuffer" type="http"
  ondata="this.handleData(this);"
  onerror="this.handleError(this);"
  ontimeout="this.handleTimeout(this);">
</dataset>
<dataset name="dsProducts"/> ← Specifies the destination dataset
```

**Specifies the
buffering dataset**

The HTTP standard specifies that any response status containing a numeric value between 200 and 299 is considered to be valid, resulting in a call to the `ondata` handler. A response in the 400 to 599 range is an error, resulting in a call to the

onerror handler. An example of a typical error response is the 404 error generated when an HTML page can't be found. 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.

16.1.3 Pooling buffering datasets

The `LzHttpDatasetPool` service maintains a pool of buffering datasets for sharing among any number of destination datasets. Datasets are not preallocated 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. This call obtains a buffering dataset:

```
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 associates a method 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.

16.1.4 Building a data service

A *data service* is a global object providing an HTTP-based service that is easily accessible by other Laszlo objects. For example, a useful place for a data service is the use of `getProducts` to populate the Product List window in the Laszlo Market. A data service isolates lower-level HTTP-based communications within a convenient interface.

HTTP server communications are built on requests and responses. We'll want to keep the server implementation isolated in a class, for easy revising to work with other networking technologies such as SOAP or XML-RPC. Instead of directly invoking the methods of a dataset, we'll create a data service object to encapsulate the HTTP communication details for the dataset.

A data service interface consists of a matching pair 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 behind the interface contains the common data transfer operations for buffered datasets.

Let's suppose that we have a `dsProducts` dataset contained in a `product-DataService` object whose request method is called `getProducts`. The `productDataService`'s `getProducts` method, used to populate the `dsProducts` dataset, looks like this:

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

The advantage of data service objects is that they can easily be integrated into an application. Suppose we are controlling an application with a state controller. We could control the populating of a product listing for a particular screen by adding a call to the `productDataService`'s `getProducts` method (see listing 16.2).

Listing 16.2 Sample application state controller

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

Listing 16.3 shows a supporting `gDataService` library to handle the buffering dataset issues required to handle HTTP requests and responses. 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 an error indicator.

Listing 16.3 `productDataService.lzx` handles server-related interaction for the `dsProducts` dataset

```
<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>
```

Listing 16.4 shows a `getProductParams` method for packaging the URL parameters into an associative array. Although Laszlo provides an `LzParams` utility for handling HTTP parameters, a JavaScript associative array works just as well.

Listing 16.4 `productDataService.lzx`: building a query string

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

Listing 16.5 shows 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 adopt a convention of using the request name and appending `Result`.

Listing 16.5 `productDataService.lzx`: getting a product listing

```
<method name="getProducts">
  var requesturl = pathurl + "products.do";
  var params = getProductParams();
```

```

    gDataService.sendRequest(this,
        requesturl, params,
        "getProductsResult");
</method>
<method name="getProductsResult" args="status, data">
    if (status == true)
        this.appendChild(
            data.getFirstChild());
    else
        Debug.write("getProductsResult Failed: " + data);
</method>

```

Establishes request-response link

Updates 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

For a clean interface to our data services, we package the common underlying methods into a globally accessible node-based object called `gDataService`, which

- Obtains and disposes of a buffering dataset
- Transfers data from the buffering to the destination dataset
- Provides an asynchronous response method
- Provides standard error handling

To make it globally accessible, we define the `gDataService` object as a top-level variable in a library in listing 16.6. It uses the `LzHttpDatasetPool` service with a 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 16.6 `gDataService.lzx`: encapsulating common HTTP functionality

```

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

```

Sets up data, error, timeout delegates

```

<method name="sendRequest" args="sender, url,
    params, response">
    var ds = LzHttpDatasetPool.get(this.loadDel, this.errorDel,
        this.timeoutDel);
    ds.setAttribute("sender", sender);
    ds.setAttribute("response", response);
    ds.setSrc(url);
    ds.setQueryType('POST');
    ds.setQueryParams(params);
    ds.doRequest();
</method>
...
</node>
...
</library>

```

Saves sender name, response method

Updates HTTP settings

Sends 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 `error` and `message` attributes containing an error description:

```

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

```

This approach, shown in listing 16.7, centralizes error processing and simplifies the response method since it is now guaranteed to receive only valid data.

Listing 16.7 `gDataService.lzx` (cont): associating the response with the request

```

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

```

1 Finds status code

2 Checks error attribute

3 Displays error message

4 Invokes response handler

The method `dsLoad` first checks the status code ❶ by searching through the first child node for a status node. If a status node ❷ is found, its error attribute is checked and its message attribute ❸ is displayed in an error window. Remember that `getAttr` retrieves XML attributes, while `setAttribute` sets dataset attributes. Next, we invoke the response handler ❹. 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 are 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. Because 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 `getProducts-DataResult` 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 (see listing 16.8).

Listing 16.8 `gDataService.lzx`: handling error and timeout conditions

```
<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 error messages are automatically displayed. This provides a wide latitude for how these messages are displayed. Now that you've seen how to communicate with an HTTP server, let's use it to save our application state.

16.2 Accessing sessioned data

Although a Laszlo application can operate independently by maintaining state in its datasets, this data is released and lost when the application terminates. Longer persistence requires an outside server to store its state. Since HTTP is a stateless protocol, a server uses a *session* object to maintain state for each client. A session generally has a time limit, 30 days for example, before it needs reinitializing.

Associating a server's session with a particular browser requires a way to associate it with a particular client browser. Each session contains a session *id* that allows it to be easily located. Including this session *id* with each request from a client browser identifies its corresponding session state. Several techniques have been developed to transfer a session *id*: cookies, URL encoding, and SSL sessions. Cookies are the most popular method, so we'll look at them. A web server initially inserts a cookie containing a unique session identifier into a response header and sends it to the browser. Assuming that the browser supports cookies, it processes the cookie header and stores it for later use. On subsequent requests, the browser includes this cookie. When the server processes a request, it checks the cookie for the session *id* to identify the browser.

Laszlo uses this same mechanism for both its Flash and DHTML applications. Because the browser automatically performs the processing, Laszlo doesn't have to perform any special processing. Everything works just as with regular HTML applications. Although Flash has its own session mechanisms such as Flash cookies, there's no reason not to use the cookie facilities available in the browser.

16.2.1 Building a sessioned shopping cart

Whenever a shopping cart's contents are updated, we call the shopping cart data service to update the server's session to reflect the changes. This data service contains the CRUD (create, replace, update, and delete) methods to update a session's contents to reflect these actions:

- Populating a shopping cart from a session
- Updating a sessioned shopping
- Deleting from the shopping cart

The following sections examine each of these operations.

Populating a shopping cart from a session

At application startup, the shopping cart needs to be initialized with any previously saved contents. These contents are obtained through the `cartDataService`'s

getShopCart method. In listing 16.9, this method is added to the state controller's Login to Main state to ensure the shopping cart window is updated before the main screen is first displayed.

Listing 16.9 controller.lzx: retrieving the initial shopping cart contents at startup

```
<library>
  <node id="gController">
    ...
    <handler name="onappstate" args="state">
      ...
      case "Login to Main":
        title = "Login to Main";
        productService.getProducts();
        cartDataService.getShopCart();
        break;
    </handler>
  </node>
</library>
```

Listing 16.10 shows that when valid data is received by the response method, it is copied to the dsCart dataset. Since this data conforms to the shopping cart object's data path XPath expressions, the shopping cart's display is automatically updated to reflect the contents. Afterward, the total amount for the shopping cart items can be calculated by the shopping cart object's updateTotals method.

Listing 16.10 cartDataService.lzx: retrieving initial contents

```
<library>
  <dataset name="dsCart"/>

  <node name="cartDataService">
    <attribute name="pathurl"
      value="$once{canvas.apiurl + '/store/'}"
      type="string"/>
    <method name="getShopCart"> ← Builds request for cart contents
      var params = null;
      var requesturl = pathurl + "xcart";
      gDataservice.doRequest(this, requesturl, params,
        "getShopCartResult");
    </method>

    <method name="getShopCartResult" ← Receives cart contents
      args="status, data">
      if (status == true) {
        dsCart.setChildNodes([data.getFirstChild()]);
        main.shoppingcart.shopcart.

```

```

        updateTotals();
        Debug.write("ShopCartData returned: ", data);
    } else {
        Debug.write("getShopCartDataFailed: "+data);
    }
}
</method>
...
</node>
</library>

```

← Calculates shopping cart totals

Now that we know how to populate a shopping cart, let's look at the other CRUD-related methods supported by the cart data service to manage a sessioned shopping cart.

Updating a sessioned shopping cart

In chapter 12, we added a set of scoreboarding methods to enable all input sources to easily update the shopping cart. One of these methods, `updateShopcart`, now needs to be updated from operating with a local dataset to communicating its changes to an HTTP server. Because adding and updating a shopping cart item both occur in this method, we'll handle them together. In either case, the server must update its stored session. Listing 16.11 shows `updateShopcart` updated with additional persistence-related methods.

Listing 16.11 `shoppingcart.lzx`: telling the server to add or update an item to a session

```

<class name="shoppingcart" ... >
...
<method name="updateShopcart" args="dp">
    <![CDATA[
        var curr = dp.xpathQuery("@sku");
        var exist = dptr.xpathQuery("item/@sku");
        if (exist != null) {
            if (typeof exist != "object") {
                var items = new Array();
                items[0] = exist;
                exist = items; }
            for (i = 0; i < exist.length; i++) {
                if (exist[i] == curr) {
                    dptr.setXPath("dsCart:/items/item[@sku='"
                        + curr + "']");
                    var qty = dptr.getNodeAttribute("qty");
                    var sku = dptr.getNodeAttribute("sku");
                    dptr.setNodeAttribute("qty", ++qty);
                    cartDataService.
                        updateShopCartItem(sku, qty);
                    main.shoppingcart.shopcart.update_totals();
                    return; }}}
    ]>

```

① Updates quantity of existing item

```

        var ele = new LzDataElement("item");
        var sku =
            dp.getNodeAttribute("sku");
        var title =
            dp.getNodeAttribute("title");
        var image =
            dp.getNodeAttribute("image");
        var price =
            dp.getNodeAttribute("price");
        ele.setAttr("sku", sku);
        dptr.p.appendChild(ele);
        cartDataService.addShopCartItem(sku);
        main.shoppingcart.shopcart.update_totals();
        return;
    }>
</method>
</class>

```

② Creates new item for cart

③ Adds new item to cart

When a matching SKU isn't found in the shopping cart, a new item is created ② with a default quantity of 1. The `addShopCartItem` method updates the session ③ with this new item. When a matching SKU is found, only its quantity ① is updated. The SKU is used to find the matching item in the stored session.

To support persistence in the shopping cart, we only need to add calls to the cart data service's `updateShopCartItem` and `addShopCartItem` methods. Listing 16.12 shows the implementation of these methods.

Listing 16.12 `cartDataService.lzx`: adding and updating items

```

<method name="addShopCartItem" args="sku">
    <![CDATA[
        var requesturl=pathurl + "add_to_cart/";
        var params=this.getShopCartParams(sku,1);
        gDataservice.doRequest(this, requesturl,
            params, "getStatusResult");
    ]]>
</method>
<method name="updateShopCartItem" args="sku, qty">
    <![CDATA[
        var requesturl = pathurl + "update_cart/";
        var params =
            this.getShopCartParams(sku, quantity);
        gDataservice.doRequest(this, requesturl,
            params, "getStatusResult");
    ]]>
</method>
<method name="getShopCartParams"
    args="sku, qty">
    var params = {};

```

Buils request for new item

Buils request to update item

Buils parameter array

```

        params.sku = sku;
        params.qty = qty;
        return params;
    </method>
    <method name="getStatusResult"
        args="status">
        if (status == false)
            Debug.write("getShopCartDataFailed: "+data);
    </method>

```

Contains common status response

`addShopCartItem` and `updateShopCartItem` only return a status response, so we'll create a single response method called `getStatusResult` to handle them. Also, the supporting `getShopCartParams` method is used to convert arguments into URL parameters to be included in an HTTP request to the server.

Since a shopping cart's quantity field can be directly updated through its input field, this is another spot where the server's saved shopping cart must be updated with the `updateShopCartItem` method (see listing 16.13).

Listing 16.13 shoppingcart.lzx: updating the number of items

```

<edittext name="qty" valign="middle" width="30" fontstyle="bold"
    doesenter="true" fontsize="10" datapath="qty/text()">
    <method name="doEnterDown">
        var qty = this.datapath.setNodeText(this.getText());
        var sku = this.datapath.getNodeAttribute("sku");
        cartDataService.updateShopCartItem(sku, qty);
    </method>
</edittext>

```

In a complete implementation, we'd check for negative or non-numeric values, and enforce an upper limit on the item number. We omit these details here to focus on the central HTTP issues.

16.2.2 Deleting from the shopping cart

We've shown how an item can be deleted from the shopping cart by dragging and dropping it into the trash. Listing 16.14 updates this with a call to the cart data service's `deleteProduct` method to instruct the server to delete this item from its session.

Listing 16.14 main.lzx: deleting a shopping cart item when it's dropped into the trashcan

```

<handler name="onmousetrackup">
    ...
    var sku = dragger.datapath.getNodeAttribute("sku");

```

```

        cartDataService.deleteProduct(sku);
    </handler>

```

Listing 16.15 shows the `deleteProduct` and its supporting parameters method that sends a request to the server to delete this product from its session.

Listing 16.15 `cartDataService.lzx`: deleting a product

```

<method name="deleteProduct" args="sku">
  <![CDATA[
    var requesturl =
      pathurl + "delete_product/";
    var params =
      this.getDeleteProductParams(sku);
    gDataservice.doRequest(this,
      requesturl, params, "getStatusResult");
  ]]>
</method>
<method name="getDeleteProductParams"
  args="sku">
  var params = {};
  params.sku = sku;
  return params;
</method>

```

Builds HTTP request to delete item from cart

Builds parameter list

This completes the CRUD-related data services for managing the server's sessioned shopping cart. Whenever the contents of a shopping cart are modified, one of the `cartDataService` methods is called to build and send an HTTP request to the server. Although other data services, such as login and order completion, are required in a real application, we've omitted them here since they're just another set of data services.

As Figure 16.2 shows, determining the API marks the end of user-centered design. To understand how a back-end server implementation, such as Struts or Ruby on Rails, supports the Laszlo Market with XML-over-HTTP services, please take a look at appendix A or B online.

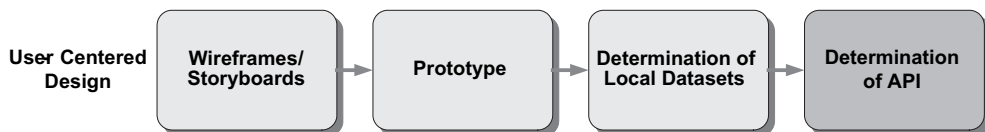


Figure 16.2 In our top-down development framework, we have advanced to the final stage of determining the API.

16.3 Maintaining server domains

Real-world application development requires an application to be deployed in different domains during its development lifecycle. An application typically moves from its initial development platform to a *staging environment* for testing and QA. After successfully completing its QA phase, an application is ready for production deployment. A *server domain* is an HTTP server designed specifically to support a particular phase of application development. For the Laszlo Market, we'll move through three phases: initial development, staging, and deployment. An application needs to be able to easily switch domains. To do this, we'll create a config.xml file to supplement the main controller.lzx file. An `url_env` attribute is added to the `gController` object to control the definition of an `apiurl` attribute containing the current deployment environment: `dev`, `staging`, or `www` (for production):

```
<library>
  <attribute name="apiurl"
    value="http://localhost:8082" type="text"/>

  <state apply="{gController.url_env == 'dev'}">
    <attribute name="apiurl" value="http://localhost:8082"
      type="text"/>
  </state>
  <state apply="{gController.url_env == 'staging'}">
    <attribute name="apiurl" value="http://localhost:8080"
      type="text"/>
  </state>
  <state apply="{gController.url_env == 'www'}">
    <attribute name="apiurl" value="http://www.laszloinaction.com"
      type="text"/>
  </state>
</library>
```

The current deployment environment is easily changed by updating the value of the `gController`'s `url_env` attribute. Instead of a static value, a browser's URL query string can be used in a constraint to dynamically set the HTTP server domain. Now this value is propagated throughout the application, updating all the URLs.

In this chapter, we've shown how a Laszlo application can easily transition from working with local datasets to network-supplied data. This approach delays integration issues to the end of application development to minimize their impact. Using the supporting classes shown in this chapter provides an easy path for this integration. This approach can also be supplemented with server domain settings, providing a migration path for an application from development to final deployment.

16.4 Summary

This chapter demonstrated the ease with which a local dataset implementation can be converted to interface to an HTTP server. Our initial code base was transitioned to a networked environment by adding a handful of data service calls that handle all HTTP-related communication with the server. Since these data services are methods in a dataset object, they update that dataset. This isolation of a Laszlo application from implementation details made the transition relatively easy.

Because we ensure that the composition of an XML document returned by the server is compliant with the reference composition established by the local dataset, none of the data path XPath statements for the view-based objects need updating. Thus, the application's operation is identical for local and networked datasets. This development strategy relaxes the coupling between the client and server development groups, allowing each to work independently. It also encourages experimentation with different data compositions and APIs without impacting the other side's efforts.

Using XML-over-HTTP as the communication medium allows Laszlo to work with any HTTP-based web server since, by definition, all HTTP servers must support this capability. In this chapter, we only explored Laszlo's role in this exchange. Appendix A and appendix B, online, feature two server-specific implementations, Java-based Struts and Ruby on Rails, creating a complete client-to-server implementation.

Using an HTTP server with database access provides a volume of data that can't be replicated using local datasets. While this provides new capabilities, it also introduces optimization issues that are dealt with in the upcoming chapters. In the next chapter, we examine data-related optimization issues. The final chapter deals with application-level optimization issues.

LASZLO in Action

Norman Klein and Max Carlson with Glenn MacEwen

If you have not seen a Laszlo-driven site before, nor one of the Laszlo demos, you are missing a wonderful experience. Laszlo sites are continuous and flowing, often implemented as single-page apps that users can navigate with ease. Laszlo is a declarative, open-source programming language and environment that compiles to Flash, DHTML (i.e., AJAX), or Java/J2ME, with other platforms coming.

Written by an expert Laszlo developer together with a company founder, **Laszlo in Action** is and will remain the authoritative resource for this new technology. It gives you solid coverage of the Laszlo LZX language and then shows in very practical and complete terms how to use it. The authors develop a working example of an online store; they illustrate how to implement a site with Struts and Ruby on Rails; they show how to provide streaming video using Red5 server; and much more.

The book is written for web developers interested in creating improved, interactive internet applications.

Norman Klein, a former consultant with Laszlo Systems, has been a software engineer for over 20 years, and has been involved in internet development since its inception. A founder of Laszlo Systems, **Max Carlson** is a frequent public speaker, and the implementor of many of the features in the current 3.X OpenLaszlo release. **Glenn MacEwen** is a technical editor and writer living in Princeton, NJ.

For more information, code samples, and to purchase an ebook visit www.manning.com/LaszloinAction

“Every OpenLaszlo developer should have a copy.”

—Ryan Stewart, Adobe

“The definitive source—I highly recommend it.”

—Robi Sen, Twin Technologies LLC

“It bridges the gap between the documentation and a practical understanding of the platform.”

—Geert Bevin, Uwyn

“... will help you quickly and efficiently develop the service-oriented UIs you'll need!”

—Matt Raible, Raible Designs, Inc.

“... opened my eyes to a new world of internet applications.”

—Andrew Glover, Stelligent

“A definitive source for rich internet application developers.”

—Edmon Begoli

Oak Ridge National Laboratory