Creating Web-Based EDI Applications with AJAX

Joseph R. McVerry

American Coders, Ltd

&

Ralph Naylor

June 28, 2006

American Coders, Ltd.
Post Office Box 97462
Raleigh, NC 27624
(919) 846-2014
www.americancoders.com

Table of Contents

Creating Web-Based EDI Applications with AJAX

Executive Summary

The usual pattern for delivering an EDI application on the web involves building a client application for data entry and a server side application for processing the data. The tools for building these applications have advanced to new levels allowing for greater productivity in building the applications, a more consistent look and feel for the applications, and greater adaptability as the underlying EDI standards evolve.

In particular, the AJAX technology has evolved to make this process much easier using the underlying X12 and EDIFACT standards to directly generate consistent user interface templates. These templates jump start the process of generating user interfaces allowing businesses to quickly generate applications that access and leverage the EDI standards as they evolve.

Introduction

EDI web applications have been typically created using a straight forward process:

1. A server side application is created that needs a dictated set of data. EDIFACT or X12 standards are used to build the application (identify what data is required for a given business communication). This server side application will typically expect the data to be in a set format and have gone through some basic validation and processing when it is gathered at the client. This leaves the server side application focusing on business logic.
2. A web client is used to gather the data or to display the data that the server side processes. The client is typically focused on presentation.
3. In a classic web application style, the server side processing may be split over more than one server program and several client pages. For example, an initial server web program sends a form to the user's browser (including any JavaScript required to perform basic validation of the data that is gathered). The user submits the data, driving a server program that is charged with gathering the information and providing more detailed validation. This program will either confirm successful data entry or drive some sort of re-display of the data that has been entered with prompts to correct problems.

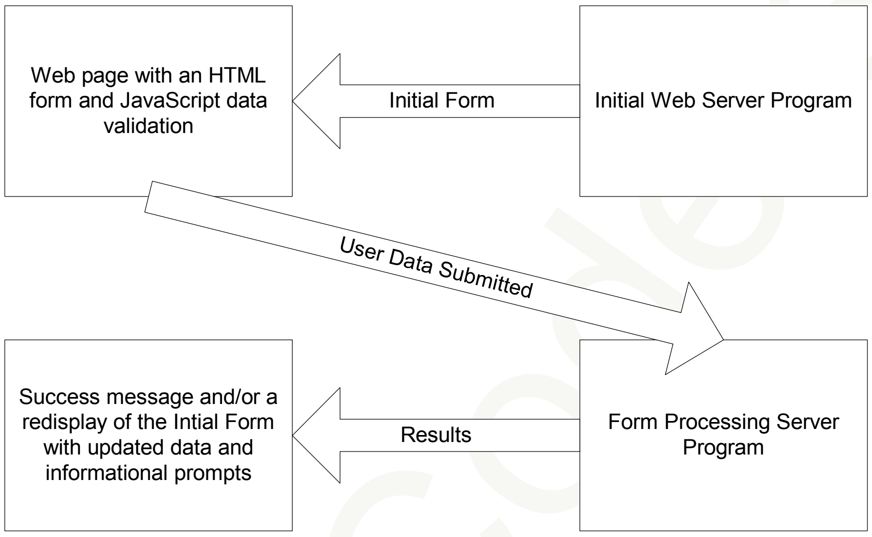

Web page with an HTML form and JavaScript data validation
Initial Form
Initial Web Server Program
User Data Submitted
Success message and/or a redisplay of the Intial Form with updated data and informational prompts
Results
Form Processing Server Program

Figure 1. A standard web application process to handle prompting for information and processing the submitted data is well known and straight forward.

This basic process seems straight forward and intuitive; but, it leads to development environments that can produce very mixed results. The structure of the server and client are created "as seems best" by developers who interpret the standards and the needs of the business. The result can be applications that do similar functions but have completely different look and feel. This is further complicated as the standards evolve requiring updates and changes in the applications.

The AJAX tool set, Asynchronous JavaScript technology and XML, has evolved within the last two years to improve the environment for developing web based applications. It does this by providing a framework that allows a web client to quickly communicate with the spawning web server application (a Java servlet). The result is that the server and client can be tightly coupled in a highly interactive environment. When this AJAX technology is coupled with XML formatted EDI messages a highly standardized application environment can be created that generates consistent EDI web applications.

The Problem:

EDI web applications are as diverse as the minds of the programmers that create them. This can be true even within the same company and development environment. They can become so divergent that two applications doing similar functions may look completely different. Managers attempting to use limited programming resource end up with divergent development environments or develop elaborate programming processes to make applications consistent. In the absence of these processes programmers may have to go through lengthy cross training before they can work on another programmer's EDI application. Even with processes and standards end users may see drastic and unwelcome changes as the applications evolve and are enhanced. The result is unnecessary expense for creation and maintenance of EDI web applications.

The Solution

A standard set of tools and programming templates can be used to build highly interactive effective EDI applications using relatively simple steps and processes. The resulting applications have a simpler architecture and a similar look and feel.

EDI standards codify standard business communications, organizing them by putting related information into structured fields, identifying required values, information only items, and expected values for the fields. In other words the basic standard has already done a lot of work in organizing the data of an EDI message. This same information organization can be used to build an application that displays and gathers the information. This application can then use dynamic html and the AJAX tool set to prompt for data entry, validate the data, and communicate it to a single server application driving a web client.

This new process retains the intuitive feel of the original web application; but, guarantees a standardized look and feel:

1. The EDI standard for a given message is first converted to XML. This XML is then input to a process that generates an EDI web based application.
2. This application prompts a user for input, validates the input, and receives the data on a server where business dependent processing can be added to the application.
3. The client uses standard DHTML, CSS, and JavaScript to present the data on the user's browser. AJAX classes are used to communicate with the Java Servlet application.

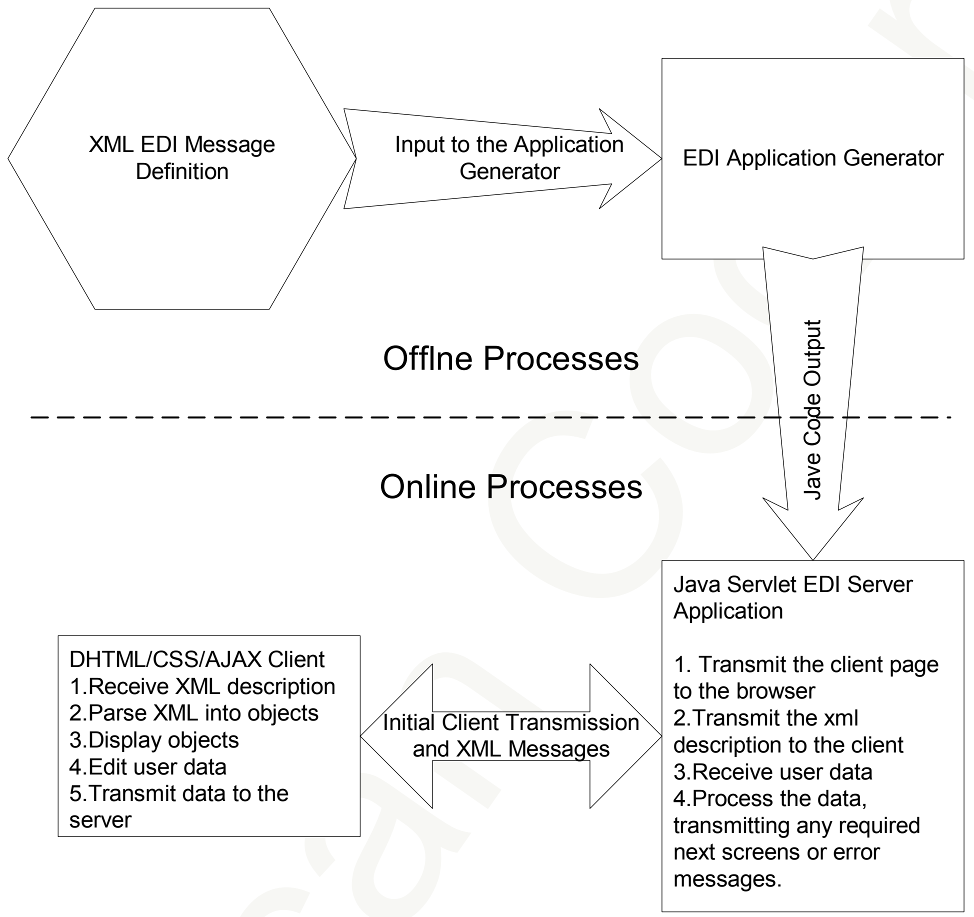

XML EDI Message Definition
Input to the Application Generator
EDI Application Generator
Jave Code Output
Offlne Processes
Online Processes
Java Servlet EDI Server Application
1. Transmit the client page to the browser
2.Transmit the xml description to the client
3.Receive user data
4.Process the data, transmitting any required next screens or error messages.
Initial Client Transmission and XML Messages
DHTML/CSS/AJAX Client
1.Receive XML description
2.Parse XML into objects
3.Display objects
4.Edit user data
5.Transmit data to the server

Figure 2. An AJAX based application can be generated from an XML definition of a standardized EDI message.

The basis of the application development is AJAX. AJAX is a collection of techniques developed by Microsoft and Google that interact with web-based JavaScript-driven applications. These applications communicate using XML (as the underlying message language) with Java servlets running on servers. Figure 2 illustrates this interaction between a web-based application and a servlet. (For another definition of AJAX visit http://en.wikipedia.org/wiki/AJAX.)

The JavaScript programs using XML, DOM methods, dynamic HTML, and CSS code create a consistent look and feel at the client. They do this because they have been generated using the structure of the EDI message with segments, composites, and elements that can be translated into a data gathering application. By using methods associated with these EDI based objects the data is displayed, edited, and collected in a consistent manner.

The following sections will show how the various EDI elements (that are defined in XML) can be used to generate the application and its various parts.

The Data Element

An individual field defined in the xml for the EDI message looks like:

> **<de pos='1' id='ST_0__1' name='Transaction Set Identifier Code' required='M' min='3' max='3' type='DisplayOnly' value='601'>**

This is an X12 EDI definition for a data element with an id of "143". The attributes "name", "required", "min", and "max" are all standard X12 constructs. There are several new attributes and changes that have been added to the XML:

1. The "id" value is different from the X12 defined value. X12 allows a segment to contain more than one element with the same id value. This is contrary to XML "best practices" which asks that the id value always be unique.
2. The "type" in this case is "DisplayOnly". In this case the application is informed that the field is not editable. The JavaScript method will make the field that is displayed non-editable.
3. The attribute "value" is not an x12 concept. "Value" is used here to allow predefined strings to be displayed as a possible value for the X12 element.

Here is another field, the date field with an id of "274" in X12 and an id of "373" is combined with the previous field:

> **<de pos='1' id='DTM_3__1" name='Requested Departure Date' required='O' min='3' max='3' type='DisplayOnly' value='274'>**
> **<de pos='2' id='DTM_3' name='Date – YYMMDD' required='M' min='6' max='6' type='DT' value='>">**

The two fields would then be displayed as:

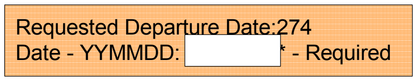

Requested Departure Date:274
Date - YYMMDD: * - Required

Figure 3. Two fields are displayed in the EDI web application.

Note that if a field is required to have data entered into as defined by "required='M'" then a prompt "*-Required" is shown to the user.

Passing the XML node named 'de' to the JavaScript DataElement constructor causes the program to build an object that can be controlled, edited, and have its value retrieved in an object-oriented fashion. That is each XML attribute becomes an attribute of the DataElement object. Here is a snipped of the constructor's Java Script code:

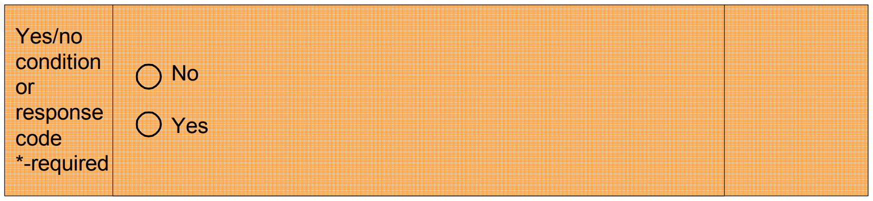
```
function DataElement(node)
{
        var attr=node.attributes
for (var i=1;i<attr.lenght;i++) {
        if(attr[i].name=='name')
                this.name=attr[i].value;
        if(attr[i]name=='pos)
                this.pos=attr[i].value
        .
        .
        .
```

Each XML attribute is stored as a JavaScript object-attribute. These objects are used to display, edit, and extract data from the web application and sent back to the server. The constructor uses the DOM methods to build an array of attributes from the XML node. The constructor iterates through the array storing individual attribute values.

List of Values

A common construct in any data input application is a "list of values". Here the user is allowed to choose between several possible values. The XML to define the YNQ segment of X12 contains an element that allows for sever possible values (Y-yes, N-no, W-not applicable, and U-unknown). In the example below only two codes of the possible four codes are passed in the XML. Also passed is the description or value associated with each code. The application will allow the user to select the field's coded value instead of having instead of having the user type in the field's possibly cryptic code value.

```
<de pos='2' id='YNQ_1__2' name='Yes/no condition or response code' required='M'
min='1' max='1' type='ID' value='>"
<valueList name='listName'>
        <id code='N' value='No'></id>
        <id code='Y' value='Yes'></id>
</valueList>
</de>
```



Yes/no condition or response code *-required

◯ No

◯ Yes

Figure 4. A list of values field is displayed as a radio button choice.

Depending on the number of items in the valueList element, the application can build the user interface to use radio buttons (as in this case), check boxes (for a single value/optional element) or lists when too many radio buttons makes the user interface difficult.

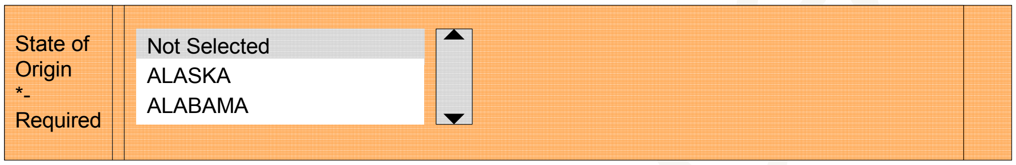


Figure 5. A list choice field is displayed.

Segments

We now move up in the XML stream structure which groups all of the data elements within a segment XML node. Like the data element a XML node is passed to a JavaScript segment constructor. The constructor extracts and stores segment attributes. Since the segment contains the data element notes its constructor is responsible for calling the DataElement constructor and stores the data element objects in an array for each data element object it contains.

```
Function Segment(node)
{
var attr=node.attributes
for (var i=0; i<attr.length;i++) }
        if(attr[i].name=='name')
                this.name=attr[I].value;
        if(attr[i].name=='id')
                this.id=attr[i].value;
        if(attr[i].name=='required')
                this.requiredattr[i].value;
}
this.dataelements=new Array();
var cd=node.childNoders;
var decent=0;
this.fieldCrossRef=Array();
for (var i=0;i<cd.length; i++) {
        if(cd.item(i).tagName=='comp'{
                c=new Composite(cd.item(i));
                this.dataelements[decnt]=c;
                decnt++;
        }
        else if(cd.item(i).tagName=='de'){
                de=new DataElement(cd.item(i));
                this.dataelements[decnt]=de;
this.fieldCDrossRef[de.pos]=decnt;
decnt++;
        }
}
```

There is a second container for data elements, composite. Since the composite is similar in function to the segment, it will not be discussed here.

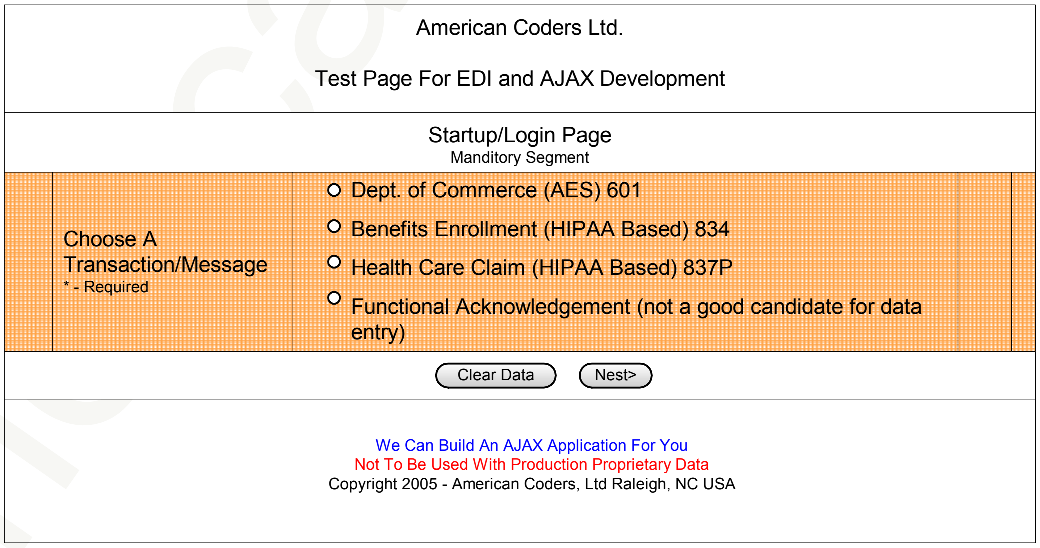
Setting Up The User Screen

Both the XML files and the EDI definitions that they represent have a hierarchical structure. This hierarchy provides a template for the application to create a consistent format when displaying user screens. As mentioned before, segments (and composites) are used to define a graphical box around all the data element data entry fields. If a composite is used then the composite becomes an inner box to the segment and the composite boxes its contained elements.

For example, in a startup application you could display choices to a user to select from a possible list of X12 messages to work on:

```
<segment id=’ST’ name=’Startup/Login Page’ required=’M’>
<de pos=’1’ id=’trans’ name=’Choose A Transaction/Message’ required=’M’ min=’3’
max=’3’ type=’ID’ value=’d’>
<valueList name=’listName’>
<id code=’601’ value=’Dept. pf Commerce(AES)601’></id>
<id code=’834’ value=’Benefits Enrollment (HIPAA Based) 834’></id>
<id code=’837’ value=’Health Care Claim (HIPAA Based) 837P’></id>
<id code=’997’ value=’Functional Acknowledgement (not a good candidate for data
entry)’></id>
</valueList>
</de>
</segment>
```

Displays as:



American Coders Ltd.

Test Page For EDI and AJAX Development

Startup/Login Page
Manditory Segment

Choose A Transaction/Message
* - Required

- Dept. of Commerce (AES) 601
- Benefits Enrollment (HIPAA Based) 834
- Health Care Claim (HIPAA Based) 837P
- Functional Acknowledgement (not a good candidate for data entry)

We Can Build An AJAX Application For You
Not To Be Used With Production Proprietary Data
Copyright 2005 - American Coders, Ltd Raleigh, NC USA

Figure 6. Segments and composites are displayed as boxes containing their contained fields.

Note: that this segment node is not the complete XML stream. The segment node is contained within a transaction node. The transaction node is discussed later in this paper.

.

Editing Data:

Editing is an important function in a data entry application. JavaScript is used to validate the user data. Basic edit rules are applied based on the requirement, data type, field length, and the value lists. Other rules such as masking and range checking can also be applied.

In X12 there are semantic rules to validate the relationship of data within a segment. By adding another XML element the application can also edit the user data using these rules.

Here's an XML node that defines the relationship of two fields in the "N1" segment:

<rule type='allOrNoneMayExist' fields='03,04'>

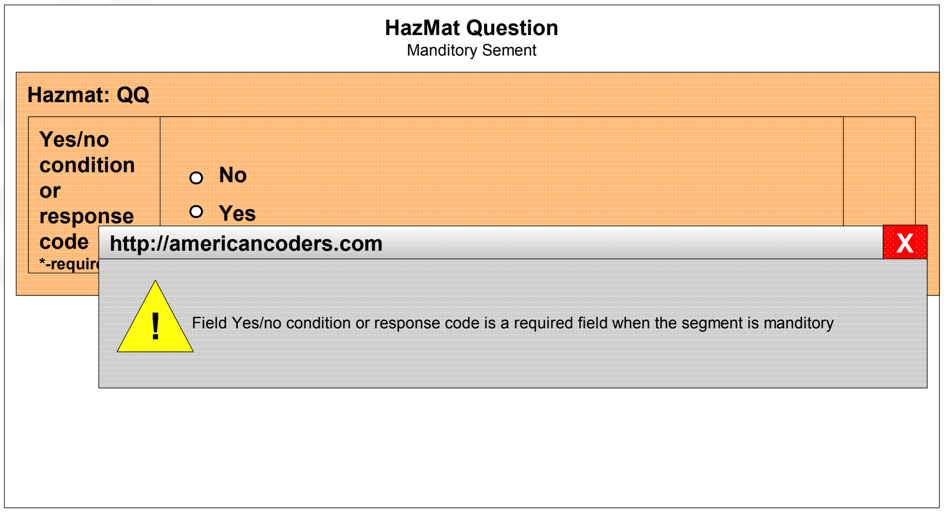
This rule states that the application should check fields N103 and N104 for two conditions:

1. Data was entered for both fields or,
2. data was entered for neither field.

(X12 experts will be more familiar with the format P0304; tis application spells out a more concise definition in its XML to aid when debugging the JavaScript program.)

Errors:

Once editing is complete the user must be informed about errors. The simplest way to do this is to use the JavaScript alert method. While the "alert" message is not cutting edge GUI methodology, an "alert" prompt can provide the user with enough information to know what the error is and how to fix it. (The newest AJAX tool kit from Google provides even more power allowing the programmer to provide more elaborate messages.)



Field Yes/no condition or response code is a required field when the segment is manditory

Figure 7. An error message is displayed using a JavaScript Alert.

Sending and Receiving Data:

Referring back to Figure 6, the user has a next button that will result in validating the data and passing it to the server if the data is correct.

Since the application and the server are not in synchronous communication, other information must be passed along with the user data. For instance, there is state data to identify the step within the overall process, an application identifier, and a time stamp. This secondary data allows the server to track what user is entering the data, what data is being entered, and where the data is to be applied and stored. The client application sends the server-required data and the user entered data to the appropriate servlet.

Below is a snippet of JavaServlet code that is used to send data to the Servlet running on the server. There are several important processes in this code. First, the program calls a system function to get a HTTP request method, XMLHttpRequest() that will be used later. (Note that this name is misleading with the XML prefix as XML has nothing directly to do with this request.) The code then sets up a handler function to respond to the message that will come back from the server. Finally the program makes an HTTP request which calls the desired Servlet. (The Servlet is waiting with a GET request.) The JavaScript then waits for a response from the Servlet.

```
if (window.XMLHttpRequest) { // get a request function for non-Microsoft browsers
xmlreq = new XMLHttpRequest();
} else if (window.ActiveXObject) { // get the ActiveX form of the function
try {
xmlreq = new ActiveXObject("Msxml2.XMLHTTP");
}
catch (actvErr) {
try { // no active x form then try the older IE function
xmlreq = new ActiveXObject("Microsoft.XMLHTTP");
}
catch (actvErr2) {
alert (actvErr2);
return ;
}
} // end of actvErr catch
} // end of get ActiveX form
req.onreadystatechange = handlerFunction;
try    {
        req.open("GET", "servletToCall'+"/?"+userData, true);
        // 1. Servlet GET request
        // 2. the Servlet name to be called along with the user data
        // 3. synchronous call.
}
catch (openErr) {
   alert(openErr)
   return;
}
var handlerFunction = getReadyStateHandler(xmlreq, incoming);
xmlreq.onreadystatechange = handlerFunction;
try { req.open("GET", prefix+theActionToTake+"/?"+sending, true); }
catch (e) { alert(e) }
```

```
return;
/**************************************************
this function simply returns an anonymous function which knows
about the xmlHTTPRequest built above and handles the response
from the server
**************************************************/
function getReadyStateHandler(req, responseXmlHandler) {
 return function () {
 if (req.readyState == 4) { // other ready state values are ignored
  if (req.status == 200) { // success
   responseXmlHandler(req); // call the fucntion to handle Servlet response
  } else {
   alert("HTTP error: "+req.status);
 }
}
}
}
```

Repeat Until Done

After the previous cycle, the whole cycle is repeated. Upon getting data back from the server, the responseXMLHandler method goes through the same steps: parsing the XML response, building the objects, displaying the data, etc.

On The Server

The JavaServlet running on the server performs three tasks:

1. It collects and stores the data.
2. It sends the user the initial screens and/or accumulated incomplete user data.
3. It maintains the current state of the process.

Of all three tasks, the collecting and storing of data is easily defined. The sending of the user screens requires some XML building skills. The collection and storing of state is probably the most interesting of the tasks. All of these will be discussed in the following sections.

Collecting/Storing User Data

Sending data to the server can be done in one of two formats:

1. an XML stream or
2. an HTTP parameter stream..

Super Class

As a GET request comes in from the client, the HTTP server applies the request to a unique Servlet class module. The primary purpose for using a super class is to track the aging of all user requests. If the request is old, as specified by either a HTTP server parameter or internally to the process the request is rejected. Otherwise the request is passed on to the child, which just happens to be the Servlet the client is working with.

Based on a “state” value in the GET requests the child Servlet calls the appropriate method to process the user data.

The Actual Servlet

If the user data is incorrect the called method returns an error message response to the client for corrective action. Back at the client this request is simple to handle. The client application does not rebuild the screen; the application simply posts an error message using the JavaScript “alert” method. If the data is good the server stores the data in a structure such as a hash table. The table is indexed by the session id assigned to the client session. This way each client’s GET request is processed by searching the hash-table using the session id, allowing a user-data container to be retrieved and have the unique data stored in the container.

State Machine

As mentioned above one of the GET request attributes contains a “state” value. This value indicates what state the Servlet expects the user to be in to process the current data. The state machine can simply be a set of Java statements to compare the state value to call the appropriate method. For example:

```
public void stateWork(String state, HttpServletRequest request, StringBuffer sb, String
timeStamp)
throws ServletException
{
this.timeStamp=timeStamp;
if (state.compareTo("init") == 0)
;//nothing to do
else if (state.compareTo("Header_ST_0_") == 0)
getHeader_ST_0_(request, sb);
else if (state.compareTo("Header_BA1_1_BA1_0_") == 0)
getHeader_BA1_1_BA1_0_(request, sb);
else if (state.compareTo("Header_BA1_1_YNQ_1_") == 0)
getHeader_BA1_1_YNQ_1_(request, sb);
/* a lot more code goes here …
*/
else
throw new ServletException("lost in state machine at input state for " + state);
```

Above, the state value was already been pulled from the GET request and passed as a String object along with a StringBuffer object and a timestamp object. The StringBuffer is used to return the response back to the user which will either be an error XML stream or the next screen’s XML stream. The timestamp, along with the session id, is used to track the user data in the hash-table.

Once the appropriate method is called, the program pulls the user data out the request stream, populates the container, and builds the next screen.

A consistent naming convention provides an easy-to-use debugging tool. The sample code below shows this naming convention. In this case, the method is working with a segment named “BA1” within a loop, also named “BA1,” both of which are contained in the Header table.

```
private void getHeader_BA1_1_BA1_0_( HttpServletRequest request, StringBuffer sb)
throws ServletException
{
String segChanged=request.getParameter("segChanged");
Segment seg;
boolean insertSeg = false;
if (segChanged.indexOf("BA1") > -1)
  {
  sc = (SegmentContainer) loopSegmentContainerStack.peek();
  tsc = (TemplateSegmentContainer) templateLoopSegmentContainerStack.peek();
  Integer minUse = (Integer) doneTable.get("Header_BA1_1_BA1_0_");
  if (minUse != null && minUse.intValue() > 0) {
    seg = sc.getSegment("BA1");
    if (seg == null) {
      seg = initSegment(sc, "BA1");
      insertSeg=true;
    }
  doneTable.remove("Header_BA1_1_BA1_0_");
  int iMin = minUse.intValue()-1;
  if (iMin > 0) doneTable.put("Header_BA1_1_BA1_0_", new Integer(iMin));
  }
else {
    seg = initSegment(sc, "BA1");
    insertSeg=true;
}
  de = getAndSetDE(seg, 1, request.getParameter("BA1_0__1"));
  de = getAndSetDE(seg, 3, request.getParameter("BA1_0__3"));
  de = getAndSetDE(seg, 4, request.getParameter("BA1_0__4"));
/* a lot more similar code goes here …
*/
putHeader_BA1_1_YNQ_1_(sb);
}
```

The methodology to process the user data is straight-forward:

1. The application retrieves the HTTP GET request.
2. The application determines if any data is to be processed.
3. Then the application gets the containers for the data.
4. The application populates the data into the container.
5. The application calls a method to build the next screen.

Of course more needs to be done, such as exception handling, internal data passing, more data editing and repetition control logic.

When the client JavaScript collected data it also set an indicator to show what segment was changed. The method tests to see if the changed segment is applicable to this method or if should continue onto the next segment. If no segment related to the method is changed, then the method goes to its last step which is a call to another method that builds the next screen.
The current method gets the containers used by the segment from the state table and any other secondary container. Using the state table, the application also keeps track of the number of times a segment is used or if the segment has been used at all. This is accomplished using a Java Integer object. The Integer object allows the method to determine if the segment is new and if so the method creates a new segment container.

Finally the method populates the segment fields from the HTTP stream.

The method could do some further testing to find any errors that were not caught by the JavaServlet. If this is the case the method creates an ERROR XML stream, stores the XML in the passed StringBuffer and exits.

The last step calls the method that will build the user screen for the next segment. The Java code below shows this for the "YNQ" segment which is part of the BA1 loop (which itself is part of the X12 601 message). In some other situations other logic is needed such as testing to see if the first segment of a loop is present or not. In these situations if the first segment of a loop is not used then the program wouldn't simply call the code to build the screen for the next segment in the loop; but, it would call the code to build the screen for the next sibling within the current container – which is either a loop or a table.

Sending the next screen

This method simply builds the XML stream and stores the stream in the StringBuffer object being passed around. An alternate way would be to build the XML with DOM or JDOM objects.

Benefits

The use of AJAX allows a web application structure to be greatly simplified, concentrating the server application into one object oriented servlet. In the EDI environment the structured nature of the EDI message and the evolved standards for these messages can be used to generate and application when these messages are converted to XML. The result is a standards based application with a consistent user interface and a greatly shortened development cycle.

Note that the methods described here could be applied to any message structure that can be organized into XML. As a quick example, consider a web survey process. Questions can be grouped and organized by type (e.g. free from text, multiple answer, multiple choice, etc.). A standard application generator could then be used to build the survey application that would display the survey to a user community, gather their answers, and store them in a data base for reporting and analysis.

Conclusion:

At American Coders we can build these highly interactive EDI applications using AJAX technology and our EDI package OBOE. If you want to try out the application written above go to http://americancoders.com/oboeajax.html. To find out more about our programs and services, visit us at http://www.americancoders.com or call us at (919) 846-2014.