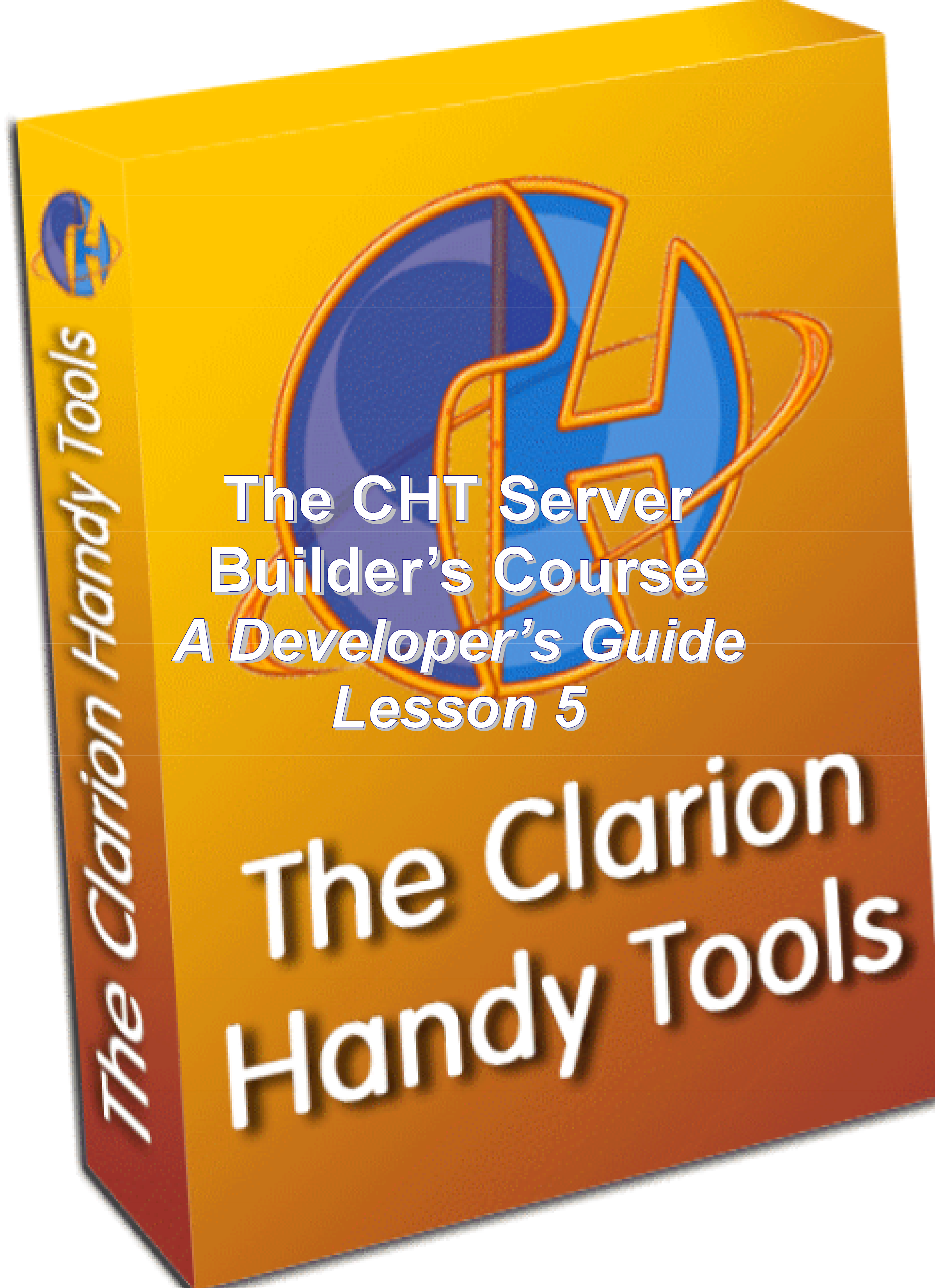

The Clarion Handy Tools
The CHT Server
Builder's Course
A Developer's Guide
Lesson 5
The Clarion
Handy Tools

INTRODUCTION

This is the last lesson in this series dealing with Server Script Concepts. In this series of script-oriented lessons, we have pointed out some key things about CHT 3rd Generation Browser Server Technology.

Here, in brief points are some of the concepts covered so far:

- CHT Browser Server Technology uses HTTP protocol
- HTTP Servers are stateless and very nearly connectionless
- Statelessness promotes a server's ability to handle multiple client connections simultaneously
- Connectionless (ness) promotes a server's ability to handle multiple clients simultaneously
- HTTP servers being stateless, store no state information about a client - with the exception of identity information when the server is a secure server that requires login
- Client state information is embedded in every Generation 3 Browser Server page
- Generation 3 Browser Server promotes the creation of dynamic-page websites as opposed to static-page websites
- Generation 3 Browser Server does not wrap HTML around back-end data - Generations 1 and 2 do wrap HTML
- Generation 3 Browser Server totally separates Data, Page Layout and Page Design
- Separation of data from page layout and design allows run-time redesign of your interactive web pages
- CHT provides source code for a design tool with which to edit HTML scripts, JavaScripts, email scripts and cascading style sheets
- CHT Browser Server Technology totally eliminates the need for back end (server-side) scripting such as ADO, PHP, CGI, PSP and others
- CHT Browser Server Technology can utilize ISAM or SQL data tables, since the server is a Clarion Application
- CHT Browser Server Technology minimizes the amount of data transfer required to render dynamic pages by a factor of 200 to 500 percent - compared to many server-side scripting techniques
- Generation 3 Browser Server data is transferred to the server in the form of JavaScript data objects
- Generation 3 Browser Server scripts are transferred to the client browser only once after any script change or after the client browser cache is cleared, otherwise dynamic page interactions transfer only data packages

In this final, script-oriented lesson, before we get into server construction in lesson 6, we will cover answers to the exercises provided with Lesson 4.

This lesson's objective is to have you understand and internalize the workings of a very important concept: **WEB FORMS**.

Web forms are the means by which web pages carry on two-way conversations with HTTP servers. Two-way interactions between browsers and servers are the basis by which browsers can become generic data client appliances for data delivery and update across the internet or intranet.

Before we begin this lesson's content, first a review of lesson 4 exercises.

REVIEW LESSON 4 EXERCISES

EXERCISE 1:
The sig.readonly property is used in 5 server scripts.
a) Name them and explain how you found this out and how long it took you.

The **sig.readonly** property is used in the following scripts:

1. (FORM) 08. Messages View/Reply Form HTML
2. (FORM) 05. Members View/Edit Form HTML
3. (FORM) 04. Members Email Form HTML
4. (JSLINKS) Write Common Menu Links Script
5. (FORM) 07. Messages View/Edit Form HTML

To determine where this property is used in the scripts do as follows:

- Click the editor, script-container area entitled **HTML/JavaScript/Mail/Style sheet**. This tells the search feature that you are going to search in the editor script area, not in one of the other fields.
- Next, click the "Find" icon, or simultaneously strike ALT/? That is, Alt, plus the question mark key on your keyboard.
- Enter **sig.readonly** and click the Global option. The Global setting ensures all scripts are searched.
- Click Ok. A list of the script names appears, containing **sig.readonly**.
- Click the list to move to the script area where **sig.readonly** appears in that script.
- To repeat any search, strike CTRL/? That is, Ctrl, plus the question mark key on your keyboard. The same search results will appear. At this time you can select the same script item you searched before, in which case the second instance of **sig.readonly** in that same script is located, if any. Or select another script name from the list to find the first instance of **sig.readonly** in that script

A search like this obviously takes only seconds. If it took you only seconds, congratulations, you're getting familiar with the editor.

b) Explain how this property is set to true or false.

The **sig.readonly** property is displayed in the server Members browse in the column headed **Rd**. To set any member's account to read-only click the **Rd** check box for that member.

c) Explain how this property is used in two of the five scripts (be specific)
Not a trick question, but obviously this property determines in all scripts whether the user may post messages or not. Therefore, one might expect that somewhere in the chain of events leading to an "Insert" or a "Reply", a check is made of this property.

Alternatively, the scripts can simply disable or hide "Insert" and "Reply" as necessary based on the state of **sig.readonly**. Rather than intervene in the actual chain of events involved in an "Insert" we have chosen simply to not display an insert menu in the event that the client's account is set read only.

Here is the code that does that:

```
/* THE INSERT MENU DISPLAY DEPENDS ON THE VARIABLE SYS.READONLY BEING FALSE. */
/* OTHERWISE THE MENU IS HIDDEN */
if (sig.readonly == false) {
   var insertmenu = '&loz;<a href="KWU$&sessionid=' + sys.sessionid +
          + sys.viewid   + '&editaction=' + flg.actioninsert +
          '&"> ' + menu.browseinsertmenu +  '</a>' ;
} else {
  var insertmenu  = '' ;
}
```

The full script can be found in **(JSLINKS) Write Common Menu Links Script**

In the script entitled **(FORM) 08. Messages View/Reply Form HTML** the state of **sig.readonly** is signaled to the JavaScript routine responsible for displaying the **reply** button.

```
<script language="javascript">
 jsbutton.drawreplybutton(sig.readonly,1) ;
 jsbutton.drawprintbutton(false,2);
 jsbutton.drawhelpbutton(false,3);
 jsbutton.drawcancelbutton(false,4);
</script>
```

The full script can be found in **(FORM) 08. Messages View/Reply Form** in the server scripts section called HTML Items.

Inside the JavaScript function **jsbutton.drawreplybutton()**, the value of **sig.readonly** is used in a conditional to determine whether the button should be enabled or disabled.

```
jsbutton.drawreplybutton(xdisabled,xtabindex) {
    btntext='<font color="white">' + button.insertbuttontext.slice(0,1) + '</font>' +
         button.insertbuttontext.slice(1, button.insertbuttontext.leng
   if (xdisabled == 0) {
      var rtnvar = '<button type="submit" accesskey=button.insertbutto
             id="btnreply" name="btnreply" tabindex="' + xtabindex +
             '" class="bldr_button" onClick="return primerecord()">' +
   }else{
      var rtnvar = '<button type="submit" disabled accesskey=button.in
             id="btnreply" name="btnreply" tabindex="' + xtabindex +
             '" class="bldr_button" onClick="return primerecord()">' + btntext + '</button>' ;
   }
   document.write(rtnvar) ;
}
```

The full script can be found in **(JSBUTTON) Draw Reply Button Script** in the server scripts section called JavaScript Items.

2) In the Mail section of server scripts, create an HTML promotional email to tell the other web app participants in your group (remember to include me) about your server and how to reach it. Use the Members --> Mail menu sequence to broadcast your email to this target audience.

Several good examples of promotional emails are provided in the script collection shipped with the server so we won't repeat any of them here. The easiest way to create a new mail script is to start with one of the existing ones until you get the hang of it.

To copy an existing script, do as follows:

1. Select any mail script item in the editor, third drop down from the left in the script editor.
2. Click Edit->Copy Local (or Ctrl+O) to obtain a list of mail scripts.
3. Select the first, for example, a script called **(MAIL) 01 Registration Email Pure HTML**.
4. Click **yes** to the question "Okay to insert a new copy of: (MAIL) 01 Registration Email Pure HTML".
5. An identical script appears with the item title: **(MAIL) 01 Registration Email Pure HTML (Copy)** and the item name: **mail.registrationhtml (copy)**. Both of these names must be unique so your job is to name your new script in a way that helps you identify it when you later use it to send mail without duplicating any existing names or titles.

6. Edit the script, using standard HTML convention.
7. For your own sanity, stick to the convention we've begun for you identifying all mail scripts with (MAIL) and mail.xxxx in the titles and names, respectively. Another convention, not so obvious, is to use the words **PURE HTML** in the item title. When you do that, the HTML switch in the SMTP mailer is automatically set for you ensuring that your mail item is sent as HTML not as text.

You will find in the sample mail script, something that may initially confuse you, called **Macro Server Variables** and **Macro User Fields**. Macros are replaced with their real-value equivalents when the email is sent. Put another way, this is **Email Merge** or **Individually Customized Mail**.

The Registration Email example begins with this line:

```
<b>Hi <INSERT USER FIRST NAME=REG:FIRST>,</b>
```

If your email were being sent to me this line would be resolved to **Hi Gus**, - in bold, since there are bold markers surrounding it. Another line in the email reads as follows:

```
This message is from <INSERT SYSTEM SERVER COMPANY CNF:ID=47> Server, a FREE, place to
share your thoughts and
```

If the email were coming from the demo server unaltered, this line would expand to: "This message is from My Web Group Server, a FREE ...." and so on. Our email uses an image header. Using images can be tricky if you're not clear on how images in emails are handled by most email clients.

```
<img src="<INSERT SERVER URL CNF:ID=40><INSERT HEADER IMAGE CNF:ID=110>" width="700"
height="114" border="0" align="center">
```

Here we've inserted two macros, first the URL back to our server followed by the name and path to an image as configured in the server variables file. This is a link back to your server such that when the email is opened, the required image is pulled down from your server by the email client, dropped into its web cache and displayed.

The principles involved are identical to web browsers displaying images in pages. If your server URL variable contains an IP, there is every chance the image will not display if your server uses a dynamic IP that changes frequently. We suggest you use a URL like **http://www.cwhandy.com/**, which resolves in the DNS server system to the correct IP for your server. Don't leave out the URL or your image will never display at all, since the email client won't know where to look for it.

Another way to do this is to use an **in-line image**. Here is how:

```
<img src="c:\mtsng\run\images\chthdrhom.jpg" width="700" height="115"
border="0" align="center">
```

By using the actual DOS path of the image on our server in our script, we can check the "In-Line Images" switch on the SMTP List Mailer dialog, the server bundles the image up with the email when sent, attaching it physically to the email as an integral component. At the email client end, the email and image are decoded to their original components, with images placed in the web-cache. Since there is no reference back to your server, this type of email is more likely to remain intact over time, since it does not have to refer back to an outside server source to acquire the image.

Macro Server Variables and Macro User Fields originate in the server variables file and in the subscriber or member file, allowing you to create an email in the abstract, customized to the information provided in the server configuration and member database. That is powerful stuff from a customer-service point of view.

We have already discussed this in a previous lesson, but in this server - to avoid becoming a spammer - your subscribers are able to control whether emails should go to them or not by accessing their subscriber record via the internet and setting the **Mail** field in their member record to **Allow** or **Disallow**.

Try the email on yourself now. Here are the steps to sending a single or mass email-merge to server subscribers:

1. From the server interface, click **Members** to open the server subscriber members browse
2. In the query control, enter a query such as NAME * SMITH to isolate your name or several names in the list (Hint: Use your own name it's not Smith.)
3. Highlight a single name or mark several subscribers using the standard MS Windows record marking technique
4. Click **Mail** -> **Send Config Mail Item**
5. In the drop down entitled Select Mail Item, choose the test mail item that you just created
6. When you do that, the **HTML** setting will becomes automatically set if you used the words "Pure HTML" in the item title.
7. Uncheck the **In-Line Images** switch if your embedded image uses a URL back to your server
8. Double check that the SMTP Server and From Address for correct configuration
9. Include any optional attachment files
10. Check that the Subject line is appropriate for the message conveyed
11. Click **Send**

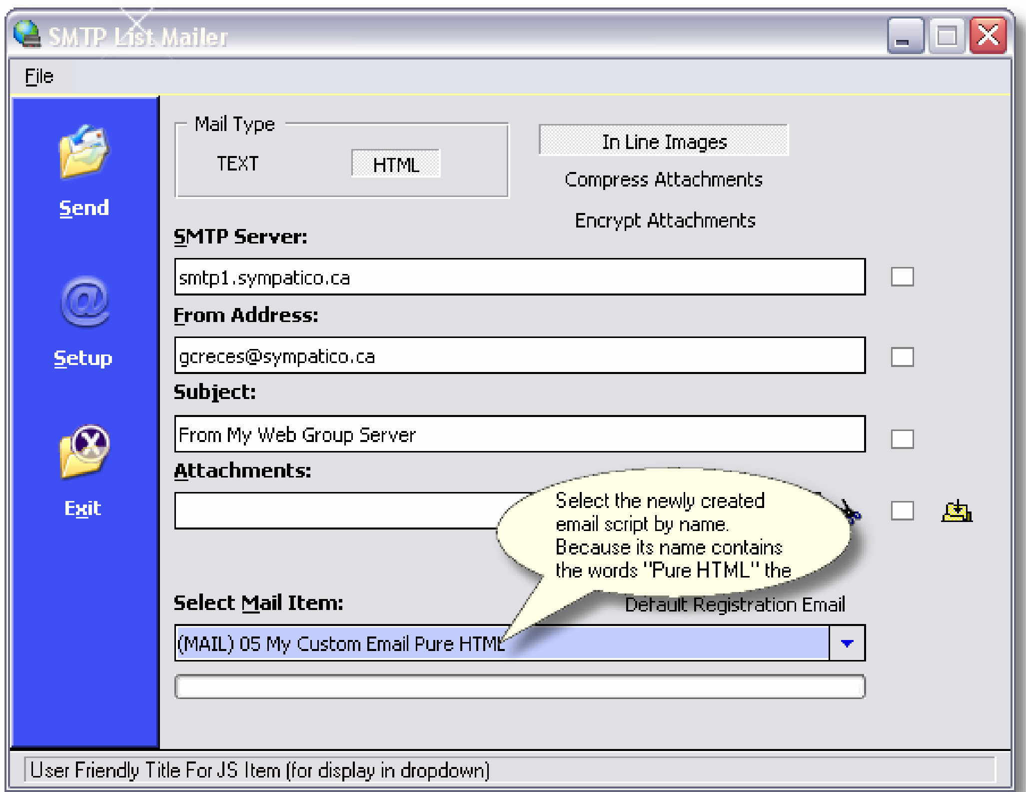


Illustration 5.1

3) In lesson 3, we had you move the contents of the title.common script to the head.image script. Add some code of your own to the title.common script to put this script container to some alternate use. Make the code conditional so it only displays on the home page.

We mentioned in the last lesson, that this script is called because the server is programmed to do this by the HTMLBuilder templates. We had you put an HTML comment in here so that it remains a legal script, nevertheless, one that does nothing. One possible alternate use for this script area, in light of the changes made in lesson three is to add something to the title bar or the header area of the page appropriate for the home page.

The code, which follows, presents the message "Welcome, Please Log In." on the home page header just underneath the company name. On the home page, we do not have access to **sig.name** the signature variable that contains the user's name, since that variable supplies a value only after the user logs in.

Here is the code we inserted:

```
<script language="javascript">
  if (sys.pageid=="home") {
     document.write('<div class="bldr_nametotop"> Welcome. Please Log In. </div>');
  }
</script>
```

To make this work correctly we modified the **bldr_nametotop** style sheet slightly to avoid overwriting the "Home Page Hits" message that displays in the left hand side of the title bar in the home page. The **left** setting now reads **80%**. In other words, display this message 80 percent of the way across the window measuring from the left.

Here's that style sheet as it appears now.

```
.bldr_nametotop  {
  font-family:                tahoma;
  font-size:                  10pt;
  font-stretch:               normal;
  font-style:                 normal;
  font-variant:               normal;
  font-weight:                bold;
  background-color:           #0088CA;
  color:                      white;
  position:                   absolute;
  top:                        126;
  left:                       80%;
  width:                      100;
  height:                     20;
  vertical-align:             middle;
  word-spacing:               normal;
  white-space:                nowrap;
}
```

The latest version of this style sheet can be found in **(Absolute) Name To Top Sheet** in the server scripts section called Style Sheet Items.

4) Research the use of the HTML <FORM> tag. Explain what it's used for and how it works on a web page. Use the knowledge gained in your research to explain what's happening in this code:

The rest of this lesson deals with the use of the HTML <FORM> tag so we'll use this exercise as a springboard into the lesson itself.

THE HTML <FORM> TAG

THE MESSAGES QUERY PAGE FORM

Form tags are the means by which web pages communicate information back to the server. Without them, web pages would be one-way from the server to the browser and never back. With <FORM></FORM>, the browser is able to collect data entry changes made on web pages and send that information back to the server. What the server does with them at that stage is a matter of server programming (your job).

```
<form action ="KQY$" method="POST" name="queryform">
    <input type="HIDDEN" id="sessionid" name="sessionid" readOnly
      value="1913-74085-7027550">
    <input type="HIDDEN" id="viewid" name="viewid" readOnly  value="NGMESSAGESVIEW">
    <input type="HIDDEN" id="editaction" name="editaction" readOnly  value="0">
    <input type="HIDDEN" id="lastquery" name="lastquery" readOnly
      value="DATE RANGE THISWEEK ORDER BY -DATE">
    <input type="HIDDEN" id="currentquery" name="currentquery" readOnly  value="">
    <input type="HIDDEN" id="defaultquery" name="defaultquery" readOnly
      value="DATE RANGE NOW ORDER BY -DATE">
    <!--------- BEGIN: Query Control written from Javascript file. ----------->
    <script language="javascript">
      document.write(unescape(form.messagesquery));
    </script>
   <!--------- END: Query Control written from Javascript file. ----------->
</form>
```

Consider the form structure we gave you as an exercise in lesson 4. The form area is enclosed between the opening <FORM> and closing </FORM> tags.

Inside the opening form tag, **action** stipulates KQY$ and the POST **method** is indicated. The form name appears after **queryform**.

Form **action** is in essence a server command. It tells the server what it is supposed to do with this data when it comes in.

Remember as we have repeated often in these lessons, HTTP servers are stateless. They do not have a clue who is talking to them at any point in time. They simply accept connections, qualify the connection for security considerations and if qualified, process the request or **action** required to complete the request. Note that the form structure contains a number of **hidden** fields followed by **javascript document.write()**, a function that writes out the messages query form.

The form being written is has the friendly name **(FORM) 06. Messages Query Form HTML**. In JavaScript code, its name is **form.messagesquery**. The whole point of this form is to enable the user enter a query which will subsequently be used to send back a browse containing specific messages data.

Another clue to the purpose of this form structure is the **action** attributed to it, namely, KQY$. If you now open HNDEQUSK.CLW in your Clarion libsrc directory you'll see KQY$ defined as **REQUEST:TakeQuery**. It's a command to the server to accept a query from the browser user.

This call writes out the **form.messagesquery** script:

```
<script language="javascript">
  document.write(unescape(form.messagesquery));
</script>
```

It creates this portion of the query page:

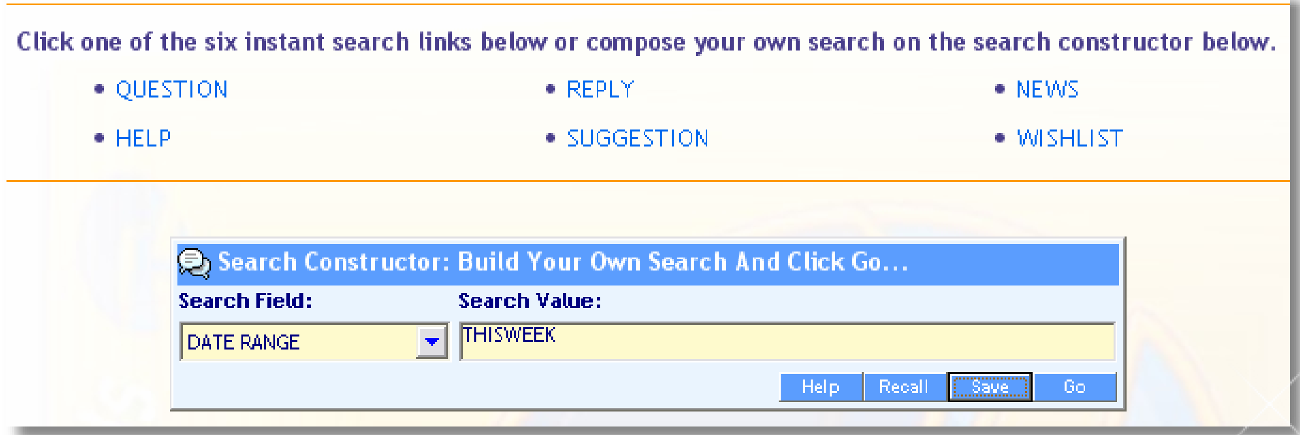


Illustration 5.2

Any one of the instant link choices or the form's GO button will trigger the form's submission to the server. Each sends a different query depending on which of these triggers is used. For QUESTION, REPLY, NEWS, HELP, SUGGESTION, HELP the queries are coded into the script itself. They are pre-set queries. In the case of the Search Constructor, the query is dynamic, based on user requirements.

Suppose we want to submit the default query, pictured here: DATE RANGE THISWEEK to the server. Clicking the GO button would do that. Here is what the server will receive, when that happens.

From the server log window here is the relevant submission:

```
sessionid=1-74163-6280940&
viewid=NGMESSAGESVIEW&
editaction=0&
queryfield=DATE+RANGE+THISWEEK+ORDER+BY+-DATE&
```

- **SessionID** - Comes back to the server from the form's **hidden** variables section. It contains state information used by the server to determine if received information should be accepted or rejected. The server uses the **sessionid** value to determine the login state of the connection. The server rejects connections when their login state cannot be determined. If logged in, the rest of the transaction proceeds.

- **ViewID** - This value is also bouncing back to the server from the form's **hidden** variable section. It's state information used by the server to determine which back end view (i.e. which Clarion process) will use the submitted query to process the request.

- **EditAction** - This flag to the server triggers it to target its return data package at a specific page script. A value of zero indicates the return of a browse-oriented data package.

- **QueryField** - This is the actual query submitted to the server. It is also a **hidden** variable. In this case, however, our web page form is actually modifying the contents of that variable based on which query triggers the submission to the server.

The server log data illustrated above is raw data coming from the browser. It is still **URL-Encoded**. URL encoding replaces spaces with plus signs. There are other URL encodings not illustrated here. Suffice to say, the server removes these and translates back to standard ASCII. The ampersands that you see here

were inserted by the browser to act as separators between the data elements (variables) being submitted, they allow the server to figure out where the information for any variable starts and stops.

THE MEMBERS QUERY PAGE FORM

Let's also take a look at the members query page and examine the <FORM></FORM> structure available there.

```
<form action ="KQY$" method="POST" name="queryform">
    <input type="HIDDEN" id="sessionid" name="sessionid" readOnly
           value="1-74166-3885085">
    <input type="HIDDEN" id="viewid" name="viewid" readOnly  value="NGMEMBERSVIEW">
    <input type="HIDDEN" id="editaction" name="editaction" readOnly  value="0">
    <input type="HIDDEN" id="lastquery" name="lastquery" readOnly  value="">
    <input type="HIDDEN" id="currentquery" name="currentquery" readOnly  value="">
    <input type="HIDDEN" id="defaultquery" name="defaultquery" readOnly
          value="DATE RANGE THISMONTH ORDER BY -DATE,-TIME">
    <!--------- BEGIN: Query Control written from Javascript file. ----------->
    <script language="javascript">
       document.write(unescape(form.membersquery));
    </script>
    <!--------- END: Query Control written from Javascript file. ----------->
</form>
```

Once again, the form **action** is KQY$, or **REQUEST:TakeQuery**. Two things are different here than in the messages query form. The **ViewID** points to a different back end view, as you would expect. It points to NGMEMBERSVIEW.

The JavaScript line that writes out the actual web query form, in this case, uses **form.membersquery**, which is titled **(FORM) 03. Members Query Form HTML** in the editor. That form looks like this when displayed in your web browser.

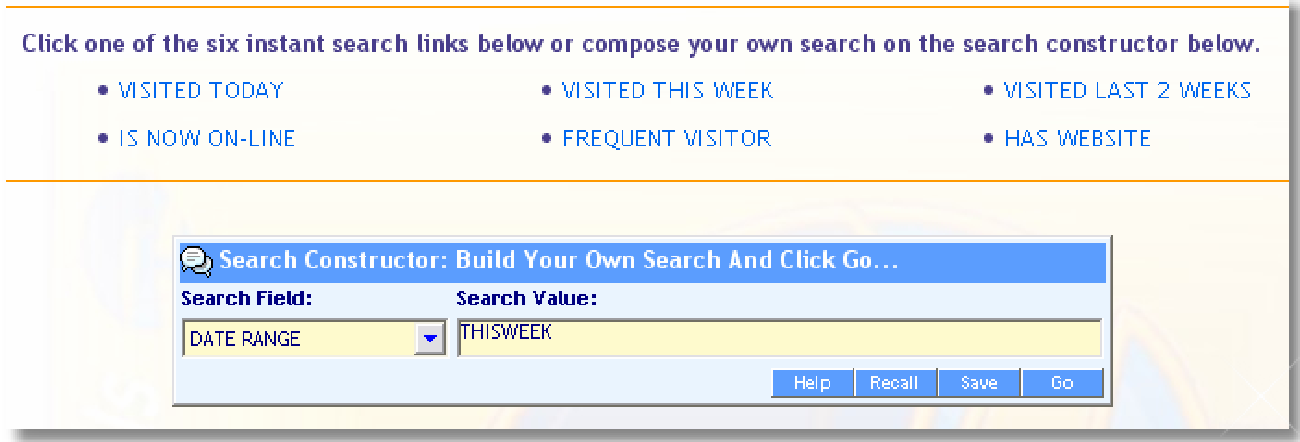


Illustration 5.3

THE MESSAGES BROWSE PAGE FORM

All pages that have data submission capabilities contain a <FORM></FORM> structure of some type. Here is the form structure used by the Messages Browse Page.

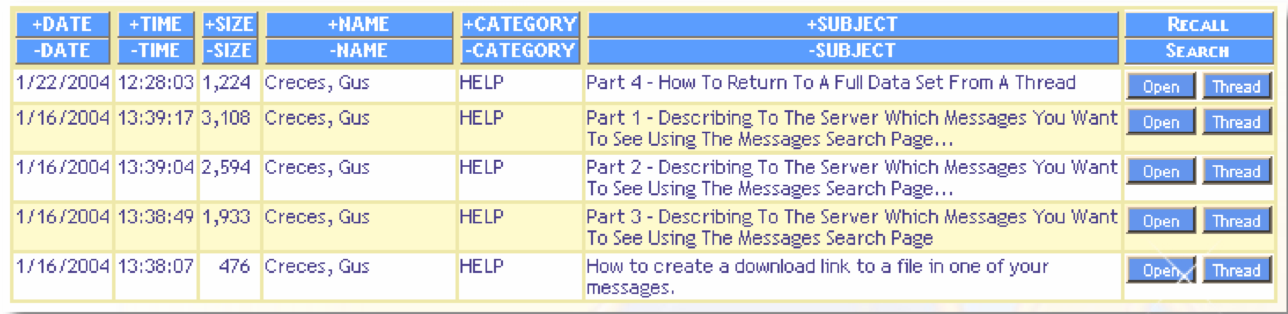
```
<form action ="KQY$" method="POST" name="ngmessagesvieweditform">
    <input type="HIDDEN" id="sessionid" name="sessionid" readOnly
      value="1-74166-3929024">
    <input type="HIDDEN" id="viewid" name="viewid" readOnly  value="NGMESSAGESVIEW">
    <input type="HIDDEN" id="editaction" name="editaction" readOnly  value="1">
    <input type="HIDDEN" id="queryfield" name="queryfield" readOnly
      value="DATE RANGE THISWEEK ORDER BY -DATE,-TIME">
    <input type="HIDDEN" id="querypage" name="querypage" readOnly  value="1">
    <!--------- BEGIN: Messages Browse Written From Javascript file. ----------->
    <script language="javascript">
          document.write(unescape(page.messagesbrowse)) ;
    </script>
    <!--------- END:   Messages Browse Written From Javascript file. ----------->
</form>
```

The action performed by the messages browse page appears to be exactly the same as the action performed by the messages and members query pages, namely, KQY$ or REQUEST:TakeQuery.

Why would that be? This is a browse page the previous two were query pages.

In fact, a web browse is simply an intermediate page between your query and the edit form. Just as a Clarion browse allows you to scan multiple records to get at the specific record or records that you want to interact with via an edit form. This browse page allows you to select the record or records that you want to read and/or edit.

When you find a message that you want to read and you click the View/Edit or View/Reply button next to it, a fresh query is spawned and sent to the server via the KQY$ action on this <FORM></FORM> form structure. That new query comes from the JavaScript data object, which represents the RECORD for any row in the browse.



+DATE -DATE	+TIME -TIME	+SIZE -SIZE	+NAME -NAME	+CATEGORY -CATEGORY	+SUBJECT -SUBJECT	RECALL SEARCH
1/22/2004	12:28:03	1,224	Creces, Gus	HELP	Part 4 - How To Return To A Full Data Set From A Thread	Open Thread
1/16/2004	13:39:17	3,108	Creces, Gus	HELP	Part 1 - Describing To The Server Which Messages You Want To See Using The Messages Search Page...	Open Thread
1/16/2004	13:39:04	2,594	Creces, Gus	HELP	Part 2 - Describing To The Server Which Messages You Want To See Using The Messages Search Page...	Open Thread
1/16/2004	13:38:49	1,933	Creces, Gus	HELP	Part 3 - Describing To The Server Which Messages You Want To See Using The Messages Search Page	Open Thread
1/16/2004	13:38:07	476	Creces, Gus	HELP	How to create a download link to a file in one of your messages.	Open Thread

Illustration 5.4

Suppose we clicked the third record from the top in the illustrated browse, the one with the subject line: "Part 2- Describing To The Server…". The JavaScript Data Object for that record looks like this:

```
function obj_brwmsg3() {
   this.bodngmemberid = "1596" ;
   this.bodupdated = "6" ;
   this.bodid = "11846" ;
   this.bodngthreadid = "11846" ;
   this.boddatelogged = "1/16/2004" ;
   this.bodtimelogged = "13:39:04" ;
```

```
   this.bodmsgsize = "2,594" ;
   this.bodname = "Creces, Gus" ;
   this.bodcategory = "HELP" ;
   this.bodsubject = "Part 2 - Describing To The Server Which Messages You Want To See Using The
   Messages Search Page..." ;
   this.fetchfilter = "BOD:ID = 11846" ;
   this.threadfilter = "BOD:NGThreadID = 11846 ORDER BY -DATE,-TIME" ;
   this.ownerrecord = "1" ;
   this.bodmessage = "<br><b>Here is another strategy for finding messages that you may not have
   thought of or considered:</b>&bull;&bull;Suppose you want to follow up on every NEW<br>More:
   (2444)..." ;
}
brwmsg[3] = new obj_brwmsg3();
```

Each of the array elements of the JavaScript data object containing the browse rows, besides containing the actual browse row information, carries with it a pre-constructed query called **obj.fetchfilter** that uniquely identifies just that record. When the user wishes to see or edit any browse record, the browse user interface must submit this pre-constructed query to the server to have just that record returned to it.

If the return record must be packaged as a browse the **editaction** sent with the KQY$ request should be **zero** or **ACTION: HTTPBrowse**. If the record should be packaged as an edit form, the **editaction** sent with the KQY$ request should be **one** or **ACTION: HTTPEdit**.

A list of KQY$ actions from **HNDEQUSK.CLW** is provided below.

```
ACTION:HttpCloseButton          EQUATE(-1)
ACTION:HttpBrowse               EQUATE(00000000B) !0
ACTION:HttpEdit                 EQUATE(00000001B) !1
ACTION:HttpChange               EQUATE(00000010B) !2
ACTION:HttpInsert               EQUATE(00000100B) !4
ACTION:HttpDelete               EQUATE(00001000B) !8
ACTION:HttpReply                EQUATE(00010000B) !16
ACTION:HttpEmail                EQUATE(00100000B) !32
ACTION:HttpRecycle              EQUATE(01000000B) !64
ACTION:HttpPrint                EQUATE(10000000B) !128
```

> **HNDEQUSK.CLW** contains many of the equate definitions used by the CHT server and internet classes.

The messages browse code attached to the **View/Edit** or **View/Reply** buttons is illustrated below. This code comes from the page entitled **(PAGE) 11. Messages Browse Page HTML,** which has the JavaScript code name: **page.messagesbrowse**.

```
/* INSERT VIEW/EDIT OR VIEW/REPLY DEPENDING IF OWNER RECORD */
if (brwmsg[datarow].ownerrecord == true) {
   buttonlabel = button.browseeditbuttontext ;
   vieweditreplybutton = '<td><button type="submit" id="btnviewedit"
                          'onclick="action=jssubmit.takeedit(brwmsg['
                          buttonlabel + '</button> ' ;
} else {
   buttonlabel = button.browsereplybuttontext ;
   vieweditreplybutton = '<td><button type="submit" id="btnviewreply
                          'onclick="action=jssubmit.takeedit(brwmsg['
                          buttonlabel + '</button> ' ;
}
```

> The latest version of the script where this portion of code belongs can always be reached in server scripts under HTML Items under **(PAGE) 11. Messages Browse Page HTML.**

A variable called **brwmsg[datarow].ownerrecord** (see browse data object above) indicates whether any message is one that was created by the logged-in individual or by someone else. When the logged in individual has created the message requested, the button text displays server variable **button.browseeditbuttontext**, which in the demo server contains **View/Edit**. When someone other than the logged in individual created the message requested, the button text displays server variable **button.browserreplybuttontext**, which says **View/Reply**.

The action attributed to both these buttons is: **jssubmit.takeedit(brwmsg[datarow])**. In other words, when this button is clicked, the function **jssubmit.takeedit()** is called and the browse data object for that given data row is passed to the function as a parameter.

```
jssubmit.takeedit(xobj){
  document.forms[0].queryfield.value = xobj.fetchfilter ;
  document.forms[0].editaction.value = 1 ;
  document.forms[0].action = "KQY$" ;
  return document.forms[0].action;
}
```

This function can be found in server scripts under JavaScript Items. Its friendly name is **(JSSUBMIT) Browse Edit Button Script.**

Here's what's happening in this function.

```
document.forms[0].queryfield.value = xobj.fetchfilter ;
```

The **fetchfilter** value from the current browse record is passed to the queryfield variable of the <FORM></FORM> structure.

This query will return one record from the browse with a query like: **BOD:ID = 11846**.

```
document.forms[0].editaction.value = 1 ;
```

The editaction variable in the <FORM></FORM> structure is being set to **one** or **ACTION:HttpEdit**. This causes the server to send back a page that requests a messages edit form rather than a messages browse.

```
document.forms[0].action = "KQY$" ;
```

This ensures that the <FORM></FORM> action is still KQY$ or REQUEST:TakeQuery and nothing else. We pointed out above, that a browse <FORM></FORM> action comes pre-set to KQY$. However, it is possible, that other subroutines could change this action to some other value for their own submission purposes, hence, this procedure simply resets it to KQY$ to ensure that the action is still correct.

```
return document.forms[0].action;
```

This causes the <FORM></FORM> structure to be submitted to the server with the established action value KQY$.

THE MESSAGES UPDATE FORM

Lets take a look now at the <FORM></FORM> structure inside a message update form like **(FORM) 07. Messages View/Edit Form HTML**.

```
<form action ="KWU$" method="POST" name="ngmessagesviewupdateform">
   <input type="HIDDEN" id="sessionid" name="sessionid" readOnly
          value="1-74166-5073527">
   <input type="HIDDEN" id="viewid" name="viewid" readOnly  value="NGMESSAGESVIEW">
   <input type="HIDDEN" id="editaction" name="editaction" readOnly  value="0">
   <input type="HIDDEN" id="queryfield" name="queryfield" readOnly  value="BOD:ID = 4">
   <input type="HIDDEN" id="bod_id" name="bod_id" value="4">
   <!---- Begin User Form Package ----->
   <script>
     javascript:document.write(unescape(form.messagesviewedit));
   </script>
   <!---- End User Form Package ----->
</form>
```

The **form action** placed on a web update form is **KWU$**. This is a server command that translates as **REQUEST:TakeWebUpdate** (see HNDEQUSK.CLW).

It tells the server to accept the contents of an update form of some type. As before, there are hidden, state variables on this form that convey what is now beginning to look like standard information, **sessionid**, **viewid**, **editaction**, **queryfield**, and a new one we've not seen before, **bod_id**, with a value of 4. This last one is actually the record identifier. Its value is used by the back end process NGMESSAGESVIEW to fetch the record with BOD:ID = 4 to which the update will be posted.

Notice there is some redundancy of information here, since the value in **queryfield** could just as easily used to isolate the target record for update. This redundancy is simply the result of design "slippage" in the HNDMTSNG.EXE demo server design - no harm done. The presence of **bod_id** *does* point out something you should be aware of when it comes to designing web forms that populate back end data. Notice that **editaction** is set to zero at this point. When this form is submitted by one of the buttons on the form, the value of this flag will be adjusted to ACTION:HttpChange, ACTION:HttpInsert or ACTION:HttpDelete, by the submission function attached to each button's **onClick** event depending on the button clicked.

JavaScript forms cannot tolerate the colon character (:) in the field name. CHT templates and web classes, in addition to setting all back end field names to lower case, they replace the colon character in BOD:ID with an underscore character which **is** legal in a JavaScript variable name.

Let's now take a look at the script for this update form. As stated, earlier, this script is called: **(FORM) 07. Messages View/Edit Form HTML**. The HTML code for this script is stored in a JavaScript variable called **form.messagesviewedit** as indicated in the **javascript:document.write()** statement above.

The message subject field in our web form is coded as follows:

```
<!---(THE OWNER UPDATE SUBJECT FIELD EDITABLE)--->
<tr>
  <td class="bldr_entry_prompt_messages" >
     <script> javascript:document.write(bod.subject.prompt); </script>
  </td>
  <td>
    <input type="text" id="Subject" name="bod_subject"
          class="bldr_edit_messages_subject" tabindex="2" >
    </input>
  </td>
</tr>
```

Two table data elements <td></td> are enclosed in a single table row element **<tr></tr>.** The first table data element is the subject field prompt. The text in this prompt comes up from the data dictionary in the data package for the record. Its name is **bod.subject.prompt**. Style sheet **bldr_entry_prompt_messages** determines prompt format.

The second table data element is the editable subject field itself. It is not simply a written-out string data as the prompt is. In fact, it is an **input** field of type **text** called **bod_subject** (for BOD:Subject).

When this edit form is submitted back to the server this variable will appear in the raw data returned as **bod_subject=Subject variable contents here.&** The server uses **bod_subject** as a tag (or token) with which to extract the contents of this variable from the data bundle submitted back with the form.

The server uses one of three functions in HNDSUBSV.CLW to perform this extraction operation. These are:

```
HNDSubscriptionServer.ExtractCommandLineItem PROCEDURE (STRING xToken)
HNDSubscriptionServer.ExtractHttpCBufferItem PROCEDURE (*CSTRING xBuffer, STRING xToken)
HNDSubscriptionServer.ExtractHttpBufferItem  PROCEDURE (STRING xToken)
```

All data returned from a web form returns with similar packaging: **TokenName=Token Data Here&** The ampersand character is a standardized means of indicating the end of a token data element. Data begins at the equal sign and ends with the ampersand. These two characters appear in our server code as:

```
HPROP:TokenValueTerminator          EQUATE('&')
HPROP:TokenValueInitiator           EQUATE('=')
```

When the critical characters like equal and ampersand are themselves data characters, the web browser sends them in "escaped" format. An escaped character consists of a % followed by the hex equivalent of the character's ASCII number. An escaped space, for instance, is %20.

Here is example code (template generated) from the HNDMTSNG.EXE sample server, extracting submitted data from the <FORM></FORM> submission:

```
CASE xServer.GetCurrentAction()
OF ACTION:HttpChange
  !Read the field contents from the web version of the
  !variable field populated on owner update, not readonly, not disabled.
  IF xServer.GetIsOwnerRecord() THEN
     BOD:Category = xServer.ExtractCommandLineItem('bod_category')
     BOD:Subject = xServer.ExtractCommandLineItem('bod_subject')
     BOD:Message = xServer.ExtractCommandLineItem('bod_message')
  END
END
```

How many fields your form populates is determined at design time by the server designer using the **BrowserServerHTMLBuilder** template.

By selecting dictionary fields on this template and indicating field state as **hidden**, **readonly**, **disabled** or **edit**, the contents of that field are sent to the server when an edit form is requested and returned from the edit form - and thus read back into the database if the **editaction** flag indicates that it should be.

The code for the browse update button comes pre-written by the server so that you can hook it to the **onClick** event of an appropriately labeled button or link. This function, as well as several other server-supplied functions is, illustrated below. They are included in the data package sent to the browser with an update request.

SaveRecord attaches to a button or link's onClick event in order to save an **existing record** back to the server.

```
function saverecord() {
   if (checkrequired() == false){
      alert("This form is causing errors. \nThere are some blank fields requiring data.\n") ;
      return false ;
   }
  if (document.forms[0] != null) {
   if ((flg.askok == flg.noask) || (confirm("Okay to save changes?"))){ ;
      document.forms[0].editaction.value = flg.actionchange ;
      return true ;
   }
  }
  return false ;
}
```

PrimeRecord attaches to a button or link's onClick event in order to save a **new record** back to the server.

```
function primerecord() {
  if (document.forms[0] != null) {
   if ((flg.askok == flg.noask) || (confirm("Okay to insert new message?"))){ ;
      document.forms[0].editaction.value = flg.actioninsert ;
      return true ;
   }
  }
  return false  ;
}
```

DeleteRecord attaches to a button or link's onClick event in order to **delete an existing** record from the server.

```
function deleterecord() {
  if (document.forms[0] != null) {
   if ((flg.askok == flg.noask) || (confirm("Okay to delete?"))){ ;
      document.forms[0].editaction.value = flg.actiondelete   ;
      return true ;
   }
  }
  return false ;
}
```

EmailRecord attaches to a button or link's onClick event in order to have the server send an email on your behalf to the individual whose record you are touching.

```
function emailrecord() {
  if (document.forms[0] != null) {
   if ((flg.askok == flg.noask) || (confirm("Okay to send this message.\n"))){ ;
      document.forms[0].editaction.value = flg.actionemail   ;
      return true ;
   }
  }
  return false ;
}
```

CancelRecord attaches to a button or link's onClick event in order to cancel update, preview or insert on any open record.

```
function cancelrecord() {
  if (document.forms[0] != null) {
   if ((flg.askok == flg.noask) || (confirm("Okay to close without changes?"))){ ;
      document.forms[0].editaction.value = flg.actionbrowse   ;
      document.forms[0].action = "RFQ$"   ;
      return true ;
   }
  }
  return false ;
}
```

The scriptwriter, to initialize the back end data to the web form variables supplied by the web form designer, may call **ConstructForm**. All data are cross-assigned by this function to the web form from the JavaScript data package containing the data for a record sent up for editing. This function throws an error when web form formatting is incorrect or the form is missing required fields expected by the server, as laid out on the **BrowserServerHTMLBuilder** template.

That feature alone simplifies form design greatly by doing all the hard stuff for you and by telling you when you have forgotten something. See forms (FORM) 04, (FORM) 05, (FORM) 07 and (FORM) 08 in your server scripts for JavaScript code examples using **constructform()**.

```
function constructform(xform) {
  if (document.forms[0] != null) {
   docform = document.forms[0] ;
   bod.category.init() ;
   bod.subject.init() ;
   bod.message.init() ;
  } ;
}
```

ClearForm attaches to a button or link's onClick event in order to clear the contents of all editable form variables.

```
function clearform() {
  if (document.forms[0] != null) {
   bod.category.me.value = "" ;
   bod.subject.me.value = "" ;
   bod.message.me.value = "" ;
  } ;
}
```

ResetForm attaches to a button or link's onClick event in order to reset the contents of all editable form variables back to their backend values.

```
function resetform() {
  if (document.forms[0] != null) {
   bod.category.me.value = bod.category.value ;
   bod.subject.me.value = bod.subject.value ;
   bod.message.me.value = bod.message.value ;
  } ;
}
```

FORM SCRIPTS IN THE DEMO SERVER

The demo server supplies eight form scripts. Not all of these update back end data. Two, already discussed here, perform query submission to the server. One sends emails to other members. One handles logins. Another manages member registration and so on.

Forms that **do** update back end data are as follows:

```
(FORM) 05. Members View/Edit Form HTML
(FORM) 07. Messages View/Edit Form HTML
(FORM) 08. Messages View/Reply/Print Form HTML
```

These form scripts are not generic. In other words, a messages view/edit form will not pass as a members view/edit form any more than a Clarion update form for your member database will serve to update a product database.

Forms are purpose-specific because they populate data base variables for display and edit. CHT templates do not generate these forms for you. A developer, in a manner not unlike Clarion's screen designer tool, designs them. One advantage you have over the Clarion approach to edit form design is that forms can be changed at run-time without recompiling your server.

Web forms are simply scripts containing a combination of HTML and JavaScript. In the HNDMTSNG.EXE demo server, when a script is changed, correctly configured client browsers feel the effects of that change immediately as soon as the change is "generated" from the script editor using the **F4** menu "Generate and Continue" or, more likely, the **Alt-X** menu "Generate and Exit". Style sheets used as illustrated in the example forms, allow you to change the colors, fonts and locations of fields on the form without touching the form script and by simply manipulating the appropriate style sheet.

DATA VIEWS IN THE DEMO SERVER

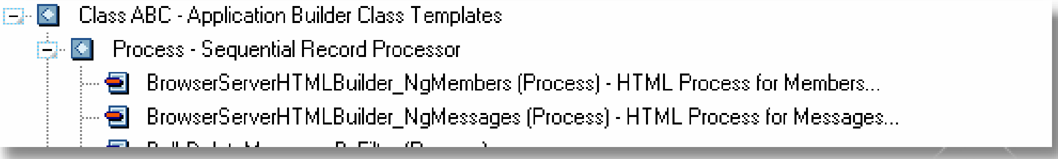


Illustration 5.5

The HNDMTSNG.EXE demo server uses two data views, each of which services a web query form, a web browse form and two web update forms. In a later lesson, we will provide you with the .APP source code for this server at which time you can examine these views for yourself. The data view procedures used are both standard ABC Process template procedures running on one or two joined databases.

The beauty of Clarion's VIEW concept is that a correctly designed view using joins, acts, or can be made to act, like a single data file. Each of these ABC process procedures has added to it a CHT template called **BrowserServerHTMLBuilder** where you decide things like which fields appear on the browse and on the form. From the dictionary, validation information, data pictures and control types passed upward to the browser through this template.

Much of the information provided by the dictionary can be changed or embellished further on the template. The design of any data view relies on your thought-through, intended purpose for that view. If a given data view does not meet all of your requirements for a given data table or combination of joined data tables, create other data views that do meet those requirements. Views are cheap and easy to design and adding more of them with different end uses in mind is the mind-set to take.

BrowserServerHTMLBuilder_NgMessages

This back end view in the demo server accepts queries from a script called **(PAGE) 05. Messages Query Page Text HTML,** which incorporates a query form element**,** called **(FORM) 06. Messages Query Form HTML** already explained earlier.

Illustration 5.6

This page plus form combination submits a query, which returns a data set displayed by a script, called **(PAGE) 11. Messages Browse Page HTML**.

A record selected from this messages browse targets one of two edit forms. These are: **(FORM) 07. Messages View/Edit Form HTML**, a form used for message insertion and change. The other form **(FORM) 08. Messages View/Reply/Print Form HTML** handles replies to other member's messages. Using two different update forms was strictly a design choice that in our view made script writing less complicated and easier to understand.

BrowserServerHTMLBuilder_NgMembers

This back end view in the demo server accepts queries from a script called **(PAGE) 04. Members Query Page Text HTML**, which incorporates a query form element, called **(FORM) 03. Members Query Form HTML** already discussed.

Illustration 5.6

This page plus form combination submits a query, which returns a data set displayed by a script, called **(PAGE) 12. Members Browse Page HTML**.

One of two forms edits a record selected from this members browse. These are: **(FORM) 05. Members View/Edit Form HTML,** a form used by members to manage their own membership record and **(FORM) 04. Members Email Form HTML,** which sends email via the server to other web group members (if they have allowed it).

In both cases, these back end views allow display or change of *only some of the back end fields* available in the view. Then only under certain conditions such as when the message is your own or the member record is your own.

View design is a data-oriented thought process much like dictionary design. The design of a back end view does not strongly dictate what the ultimate web page will look like, since web screen design is an entirely separate process performed in the Server Script Editor. Web forms may be tweaked and reworked as many times as you like to get the colors, fonts and screen positions just right. Back-end view design dictates which data fields travel to the web form and which xxxxxxxxxof those changed from the web form.

LESSON 5 SUMMATION

In these first five lessons we've talked primarily about construction of what we call the "front-end" or "client-side" of your web application. Like the tip of an iceberg, it's the part that interfaces with the world and on which your web design is primarily judged. Since from here forward this set of lessons takes a new direction towards construction of the "back-end" or "server-side" it behooves you to become intimately familiar with front-end design while you've got the chance, using the HNDMTSNG.EXE demo server. Your familiarity with what you can and can't do with client-side design and what works and doesn't work for you will ultimately make your back-end designs better and more manageable.

We've purposely kept you divorced from the source code of the back end in order to force you to think about the client-side component as an entirely separate layer of web application design and to encourage you to not confuse server-side development with client-side development. While dependent on decisions made when the server-side design is laid down, the creativity and design freedom provided by the CHT approach to web-design is well beyond what can be achieved with desk-top applications today, even using Clarion as your design tool. Your part of that equation is to develop some skills with HTML, JavaScript and CSS.

To honor that purpose and help you come to realize the possibilities, here are two exercises we hope you'll take seriously and spend time completing. The energy invested will ultimately pay dividends when the time comes that you face the design challenges provided by your own brain-children.

LESSON 5 EXERCISES

1) Design New Messages Query Form to replace the one we've given you. State the intended purpose of your design and then let other members interface with it.

2) Design New Messages Edit Form to replace the one we've given you. State the intended purpose of your design and then let other members interface with it.

We suggest you begin by making a backup of your presently workings scripts using File -> Backup Server Data and that you back up progressively as you work on your scripts. This will allow you to roll back to your last working script set using File -> Restore Server Data should you butcher a script or two along the way.

Cheers....

Gus M. Creces
The Clarion Handy Tools Page
http://news.cwhandy.ca
http://www.cwhandy.ca
support@cwhandy.com

Your index page goes here...

In MS-Word, select INDEX AND CONTENTS from the INSERT menu.
Select INDEX and click OK.