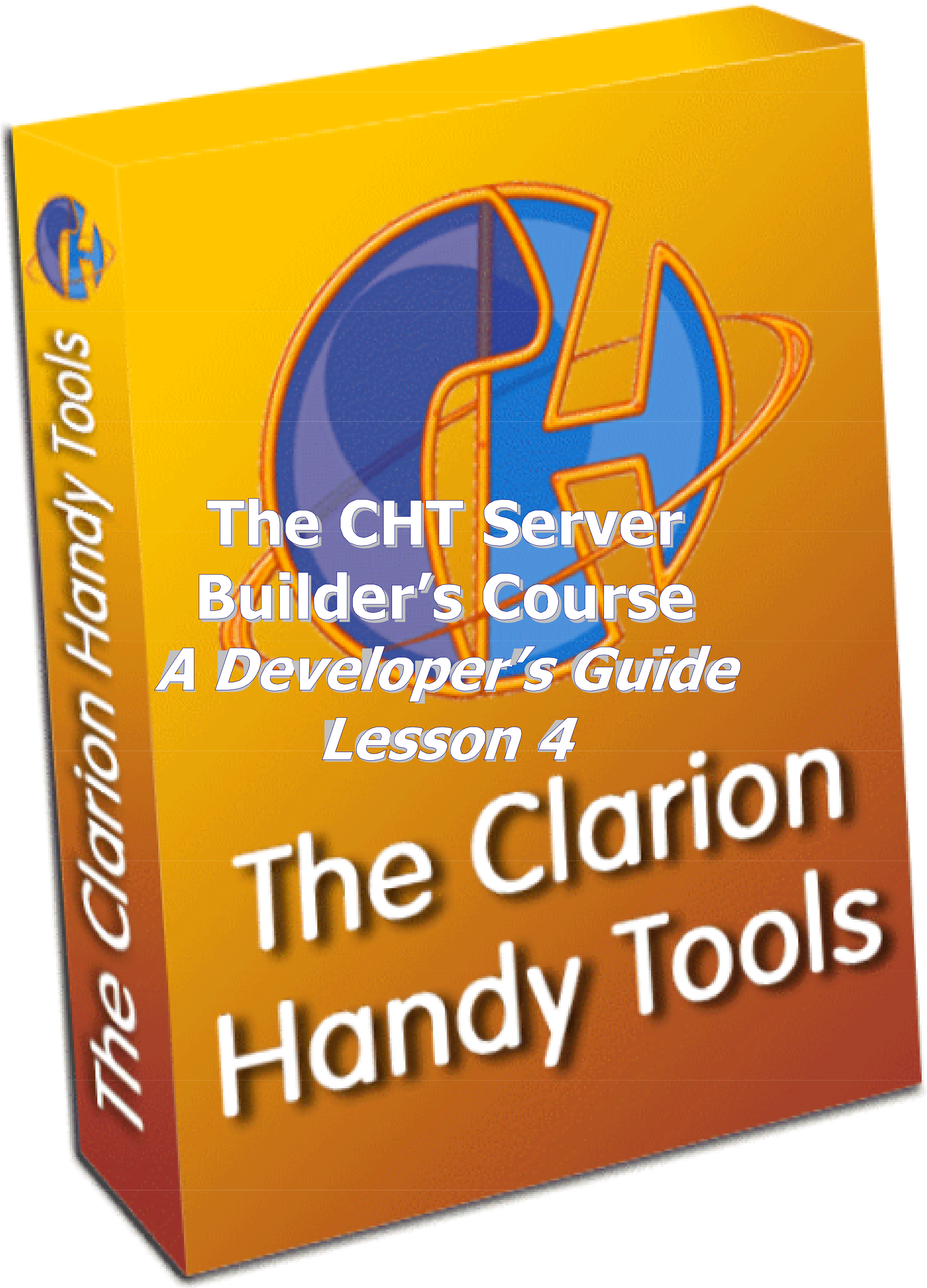


The CHT Server Builder's Course

A Developer's Guide

Lesson 4

INTRODUCTION

Keep in mind as you read this lesson that the scripts you're seeing and modifying are "client-side" scripts – scripts that are executed in the browser, at the client end of the web transaction. CHT servers don't require "server-side" scripts the way generic servers like IIS do. CHT Browser Server logic and data packaging are determined as part of your server design using CHT templates and are done with Clarion code.

Client-side scripts are used by the client appliance – namely, the browser - to render the pages at run-time, using the data provided by the server within the context of the logic designed into the server by the server developer. The design of a server, like the design of any Clarion application, is fully under your control. The server is not really involved in page-rendering at all. It only makes the page scripts, style sheets and data packages available, as needed, to the visiting browser. Final assembly is done by the browser itself.

Once you internalize this important concept, you'll begin to realize that you could, with sufficient skill and motivation, entirely re-design the HNDMTSNG.EXE server's presentation and even its intended end-use, without making a single modification to the server's Clarion source code - which you haven't been given yet.

We did some brainstorming here at The Clarion Handy Tools Page and came up with nearly a *dozen totally different end-uses* that this one server could be put to - other than acting as a Web Group server - without a single modification to the server-side Clarion code. Those possibilities will become more obvious in this lesson.

Perhaps during the course of reading and performing the tasks in this lesson, you'll have some bright ideas of your own about how to apply HNDMTSNG.EXE to a new end-use. The complete separation of data and page-rendering code and logic, as implemented in The Clarion Handy Tools 3rd generation server tools, makes this possible.

Before we delve any further into the realm of possibility, let's take a look at the reality of the assignment exercises from Lesson 3.

REVIEW LESSON 3 EXERCISES

EXERCISE 1:

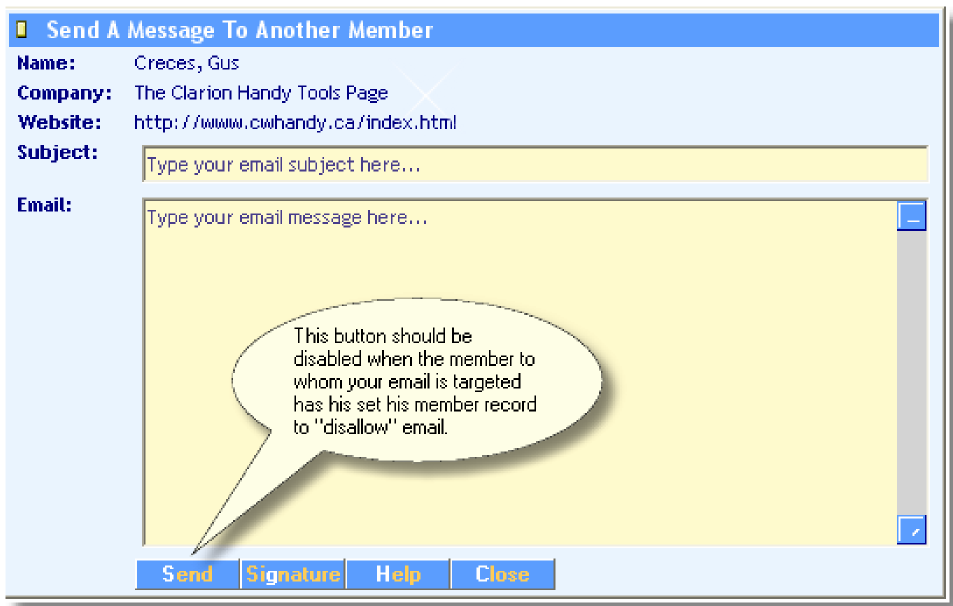
Modify the form.membersemail script to disable the email send button depending on the state of server variable reg.allowsendmail. In other words, do not allow mail to be sent via the web email form if the member has marked his member record with "disallow" in the email send field.

The **form.membersemail** script referred to in question 1 is the form that comes up when a visitor to your newsgroup clicks the "Email" button on another member's record in the members browse. This form is intended to allow a member to send email to another member without revealing the intended recipient's email address. And of course, it would not be "enlightened" if we allowed sending without giving members control over whether they actually want to receive such email or not. Member records contain a field that permits them to "allow" or "disallow" such emails. The send button should be disabled when a member has marked his personal record to disallow emails. That is the point of this first lesson 3 exercise – to modify the **form.membersemail** script to disable the send button when emailing is disallowed by the member.

The following JavaScript code creates the buttons on the bottom of the send/edit form. The button we're specifically interested in is the one drawn by a JavaScript function called **jsbutton.drawsendbutton()**.

```
<script language="javascript">
  jsbutton.drawsendbutton(sig.readonly,5);
  jsbutton.drawsignaturebutton(sig.readonly,6);
  jsbutton.drawhelpbutton(false,7);
  jsbutton.drawcancelbutton(false,8);
</script>
```

The full script can be found in **(FORM) 04. Members Email Form HTML**


Send A Message To Another Member

Name: Creces, Gus

Company: The Clarion Handy Tools Page

Website: http://www.cwhandy.ca/index.html

Subject: Type your email subject here...

Email: Type your email message here...

This button should be disabled when the member to whom your email is targeted has his set his member record to "disallow" email.
Send
Signature
Help
Close

Illustration 4.1

The script called **jsbutton.drawsendbutton()**, illustrated below, contains the source code that draws the send button. The backend variable that we're interested in reacting to is called **reg.allowsendmail**.

This is a flag in the member's registration file that he/she can set to "Allow" or "Disallow" in order to control whether the NG server sends them mail or not. This is a **reg** (or registrant) variable referring to the person whose record is being accessed.

All we need to do, to enable the send button when **reg.allowsendmail** is set to "Allow", is to test whether the "Allow" condition is being met. (See example below)

```
1. jsbutton.drawsendbutton(xdisabled,xtabindex){
2.   btntext = '<font color="white">' + button.browsesendbuttontext.slice(0,1) +
      '</font>' +
      button.browsesendbuttontext.slice(1,button.browsesendbuttontext.length) ;
3.   if (xdisabled == 0 && reg.allowsendmail == "Allow") {
4.   var rtnvar = '<button type="submit" '+
      'accesskey=button.browsesendbuttontext.slice(0,1) ' + 'i
      name="btnqsend" tabindex="' + xtabindex + '" class="bldr
      +'onClick="return emailrecord()">' + btntext + '</buttor
5. }else{
6.   var rtnvar = '<button type="submit" disabled ' +
      'accesskey=button.browsesendbuttontext.slice(0,1)' + 'id
      name="btnqsend" tabindex="' + xtabindex + '" class="bldr_button" ' +
      'onClick="return emailrecord()">' + btntext + '</button>' ;
7.    }
8.   document.write(rtnvar) ;
9. }
```

In server scripts, see JavaScript Item:
(JSBUTTON) Draw Send Button Script

The top branch of the if () statement, beginning on line 3, draws the enabled version of the button. The bottom branch, beginning on line 5, draws the disabled version. The code portion that's been expanded is **if (xdisabled == 0)**, in order to check also for the state of **reg.allowsendmail**. The if statement now reads as follows: **if (xdisabled == 0 && reg.allowsendmail == "Allow")**.

The double ampersand is JavaScript's equivalent of Clarion's AND operator. Hence, we're going to allow an enabled button (the top branch) when xdisabled is **False AND reg.allowsendmail** contains the word "**Allow**". In the JavaScript language **==** is the comparison operator, and = is the assignment operator.

Here's what the **reg** object behind this mail-send form looks like. It contains one member's registration information. It's a good example of a JavaScript data object being packaged by the server and sent to the browser as the result of a request for form information.

Note that the information available about two components, *Name* and *Company* is considerably more than the information available for say, *Website* or *Timeslogged*. That is the result of decisions made during server development about what we would be doing with the information. Some of this comes up directly from our original dictionary design. Note further, that this information is not burdened with HTML formatting tags to describe how the data should be rendered. Rendering the form or web interface is a separate operation – our script - and should not be confused or inter-woven with form data.

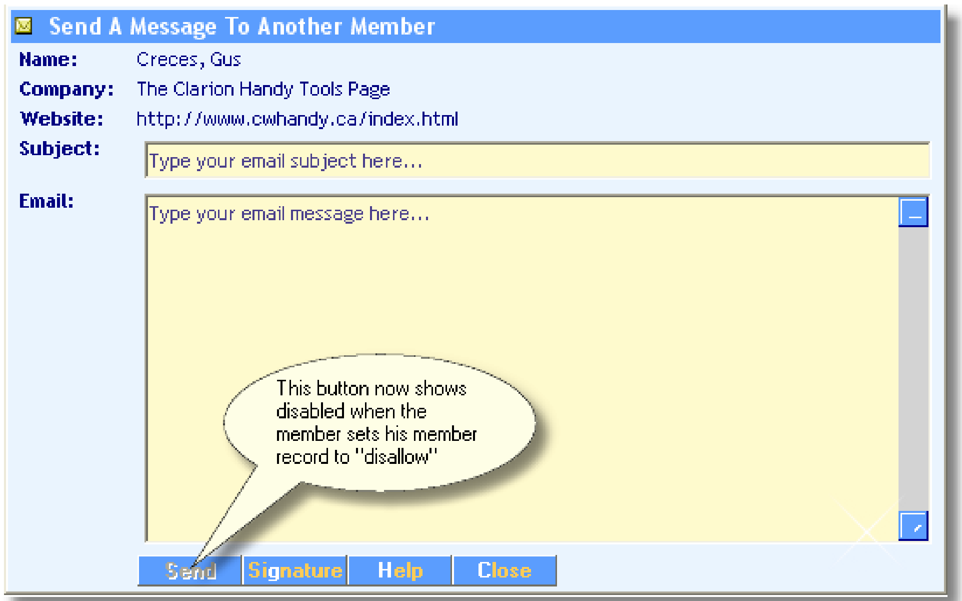
```
function obj_reg() {
   this.first = "Gus" ;
   this.last = "Creces" ;
   this.present = "Off-Line" ;
   this.datelogged = "10/29/2" ;
   this.timelogged = "11:36:1" ;
   this.name     = null ;
   this.initname = function() {
      this.me          = "";
      this.value       = unescape("Creces, Gus") ;
      this.hidden      = false ;
      this.disabled    = false ;
      this.readonly    = true ;
      this.prompt      = unescape("Name:") ;
      this.init        = function() {
        this.me        = document.forms[0].reg_name;
        this.me.value  = this.value ;
      }
   }
   this.name = new this.initname()

   this.company     = null ;
   this.initcompany = function() {
      this.me          = "";
      this.value       = unescape("The Clarion Handy Tools Page") ;
      this.hidden      = false ;
      this.disabled    = false ;
      this.readonly    = true ;
      this.prompt      = unescape("Company:") ;
      this.init        = function() {
        this.me        = document.forms[0].reg_company;
        this.me.value  = this.value ;
      }
   }
   this.company = new this.initcompany()

   this.website = "http://www.cwhandy.ca/index.html" ;
   this.timeslogged = "3559" ;
   this.allowsendmail = "Disallow" ;
   this.email = "gcreces@sympatico.ca" ;
}
var reg = new obj_reg();
```

A typical JavaScript Data Object, this member information is pure data packaged by the server, designed in the data dictionary. It is not encumbered by HTML tags or any kind of rendering information. The "client-side" scripts may use this information as required by the intended end-use.

The result of our code modification is now visible in this member's record and shows the "Send" button disabled when the member's record is set to "disallow" sending of emails.



Send A Message To Another Member

Name:	Creces, Gus
Company:	The Clarion Handy Tools Page
Website:	http://www.cwhandy.ca/index.html
Subject:	Type your email subject here...
Email:	Type your email message here...

This button now shows disabled when the member sets his member record to "disallow"
Send
Signature
Help
Close

Illustration 4.2

EXERCISE 2:
Modify the head.image script to handle not only the drawing of the page header image but also to handle some of the work now being handled by the title.common script, including drawing the title bar under the logo and the user's name at the top of the page (when known). At the same time, *do not obsolete* the title.common script so that it does nothing, since much of the tool tips text is handled also by title.common.

This is an easy one, since it's just a matter of combining the **title.common** script with the **head.image** script. Here are the two scripts as they are supplied to you.

```
<!---(The head.image script)--->
<table class="bldr_header">
   <tr>
     <td>
       <script language="Javascript">
         var headerimage = '<img src="' + image.headerimage + '
                image.headerimagewidth + '" height="' +
                image.headerimageheight +
                '" border="0" align="center">' ;
         document.write(headerimage) ;
       </script>
    </td>
  </tr>
</table>
```

In server scripts, see the HTML Item called:
(HEAD) 01. Header Image Url HTML

```
<!---(The contents of title.common moved to the head.image scri
<table  class="bldr_title_container">
  <tr class="bldr_title">
   <td>
     <script> javascript:jstitle.drawtitlebartext(); </script>
   </td>
  </tr>
</table>
```

In server scripts, see the HTML Item called:
(TITLE) 01. Common Page Titles HTML

If you copy them together just as you see them above into the **head.image** script container you're pretty much done, with one exception. In an earlier version of these scripts, that left nothing in the **title.common** script container. In that case, after completely removing the previous contents of the script container we simply added an HTML comment, which doesn't display, but is still a legal script:
<!---(THIS DOES NOTHING)--->.

Just because the basic page layout logic programmed into the server is asks for variable **title.common**, doesn't mean that variable has to actually contain anything visible.

In later versions of the **title.common** script we've added lots of other work for the script to perform and it's not necessary to pad the script with a comment to make it legal. Currently the script contains a section that puts the signed-in user's name under the page logo. We're also using this as a convenient place to store tool-tip information.

The important thing to learn from what we've done here in exercise two is that the scripts which render the look and feel of your web pages are not determined by the server at all. They can be changed, tweaked, improved, moved, on-the-fly after the server is up and running.

Admittedly, this does make the assumption that the server is sending you the correct file information for your scripts to display. But the decision about what data is sent from the server is a *data design consideration*, not an interface design consideration. Too often, web design mechanisms mix these two aspects together, making dynamic web-page design far more difficult than it has to be.

EXERCISE 3:

Find the scripts that contain and execute the actions hooked to the messages browse View/Edit button and the Thread button. Copy the script code to your answer and explain briefly what the code is doing.

The friendly name of the script to find this in is: **(PAGE) 11. Messages Browse Page HTML**. Its code name is **page.messagesbrowse**. The code that describes the buttons follows below:

```
/* 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" ' +'
   class="bldr_ebtn" '  + ' onclick="action=jssubmit.takeedit(brwmsg[' + datarow +
   '] )">' + buttonlabel + '</button>&emsp;' ;
} else {
  buttonlabel = button.browsereplybuttontext ;
  vieweditreplybutton = '<td><button type="submit" id="btnviewreply" ' +'
   class="bldr_ebtn" ' + ' onclick="action=jssubmit.takeedit(brwmsg[' + datarow +
   '] )">'  + buttonlabel + '</button>&emsp;' ;
}

threadbutton = '<button type="submit" id="btnthread" class="bldr_ebtn" ' +
   'onclick="action=jsthread.takethreadbutton(brwmsg[' + datarow + '] )">' +
   button.browsethreadbuttontext + '</button></td>' ;

/* SOME CODE OMITTED HERE FOR CLARITY */

document.write(vieweditreplybutton) ;
document.write(threadbutton) ;
```

In server scripts, see HTML Item: **(PAGE) 11. Messages Browse Page HTML.**

You already know that the browse view/edit button is only labeled "View/Edit" when the message is one that you created. Members are only allowed to edit their own messages. Otherwise the button name used is "View/Reply".

These values originate from two server variables called:
button.browsereplybuttontext and **button.browseeditbuttontext**.

The code above tests whether the property **brwmsg[datarow].ownerrecord** contains true or false. The **brwmsg[datarow]** object lets us know, via the **.ownerrecord** property, whether a message browse record belongs to the person logged in or to someone else. We'll take a further look at browse data objects later in this lesson.

What we're after here is this bit of code:
onclick="action=jssubmit.takeedit(brwmsg[datarow])"

It's the JavaScript *action* that's triggered by the View/Edit or View/Reply button's **onclick event**. Notice that the same script is called for owner message records as for non-owner message records.

The thread button doesn't change from one browse record to the next so it has no condition around its description. When the thread button is clicked, it invokes a function called: **jsthread.takethreadbutton()**. Like the **.takeedit()** function above, it is being passed the entire data record for the row being examined, that is, **brwmsg[datarow]**.

The button descriptions stored in local variables **vieweditreplybutton** and **threadbutton**, are written out as HTML in the final two **document.write()** statements: **document.write(vieweditreplybutton)** and **document.write(threadbutton)**.

Whenever you see **document.write()** you can be certain that JavaScript code is creating HTML for the browser to render.

The **jssubmit.takedit()** function looks like this:

In server scripts, see JavaScript Item: **(JSSUBMIT) Browse Edit Button Script.**

```
    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;
    }
```

Notice that the passed-in browse record is internally referenced as **xobj** (external object). The first function line performs an assignment to the document form's **queryfield** value. This is one of those hidden, form fields we discussed in the previous lesson which travels back and forth between client and server with "state" information.

We're setting **queryfield.value** to the contents of a variable called: **xobj.fetchfilter**. Though we haven't shown you yet what's in **xobj.fetchfilter**, you can guess it's a filter statement that isolates just the record you want to see when the edit form pops up. In other words, the JavaScript object containing the record for any browse data row contains a query statement provided by the server that can be used to fetch that record for editing in a form. We'll show you a browse data array later in this lesson where you can examine the data for a single browse record and the **.fetchfilter** associated with a browse record.

The hidden form field called **.editaction** is set to 1 and the page form's action is set to "**KQY$**". Look in the Clarion equates file **HNDEQUSK.CLW** and you'll find that **KQY$** is named **REQUEST:TakeQuery** and that an action value of 1 is named **ACTION:HttpEdit**. In the final line, the form's action is returned from the **.takeedit()** function such that an **onclick** of the View/Edit or View/Reply button performs the actions assigned to it inside the **.takeedit()** function.

Anthropomorphically, this **onclick** action **POST**s a message to the server as follows: "Here, take this query and return the data associated with it to me for editing or viewing."

The server wraps the data object for the requested record and sends it back with a page formation that calls for **form.messageviewedit** or **form.messagesviewreply** depending on whether you're recalling one of your own messages (for possible edit) or someone else's message (for viewing or replying).

The server, in this case decides for itself, which of the two forms to write out based on it's own determination of whether you're requesting one of your own messages or some else's.

The server page returned writes out this form for your own messages**.**

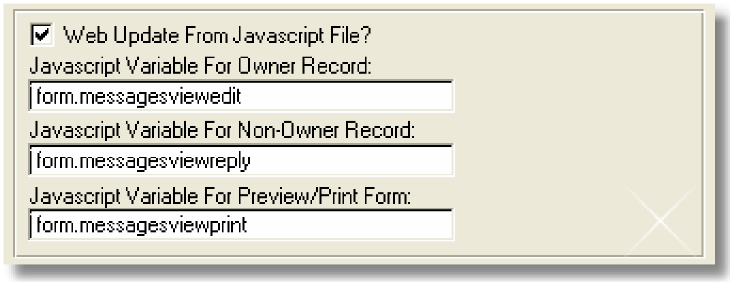
```
    <script> javascript:document.write(unescape(form.messagesviewedit)); </script>
```

The server page returned writes out this form for other member messages.

```
   <script> javascript:document.write(unescape(form.messagesviewreply)); </script>
```

The server page returned writes out this form for a print/preview version of messages.

```
   <script> javascript:document.write(unescape(form.messagesviewprint)); </script>
```



☑ Web Update From Javascript File?

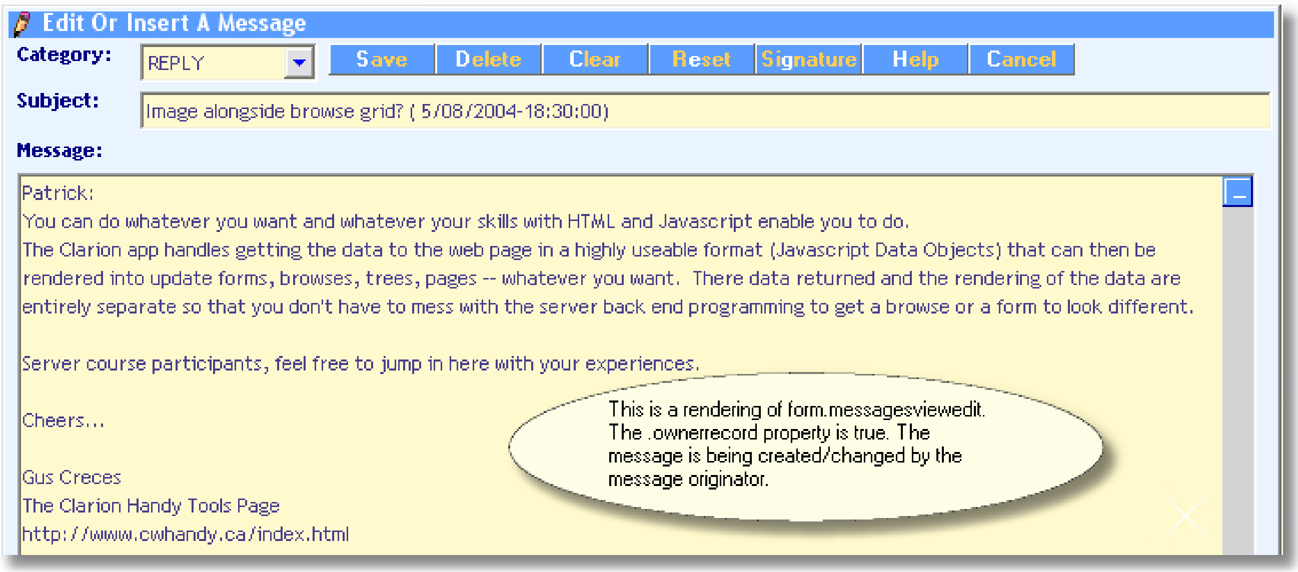
Javascript Variable For Owner Record: form.messagesviewedit

Javascript Variable For Non-Owner Record: form.messagesviewreply

Javascript Variable For Preview/Print Form: form.messagesviewprint

Illustration 4.3

Note that the latest scripts do not use all three message states provided for in the server design. Messages where the **.ownerrecord** property is true, are presented with **form.messagesviewedit**. That is, in an editable, text-format as illustrated below.



Category: REPLY

Subject: Image alongside browse grid? (5/08/2004-18:30:00)

Message:

Patrick:
You can do whatever you want and whatever your skills with HTML and Javascript enable you to do.
The Clarion app handles getting the data to the web page in a highly useable format (Javascript Data Objects) that can then be rendered into update forms, browses, trees, pages -- whatever you want. There data returned and the rendering of the data are entirely separate so that you don't have to mess with the server back end programming to get a browse or a form to look different.

Server course participants, feel free to jump in here with your experiences.

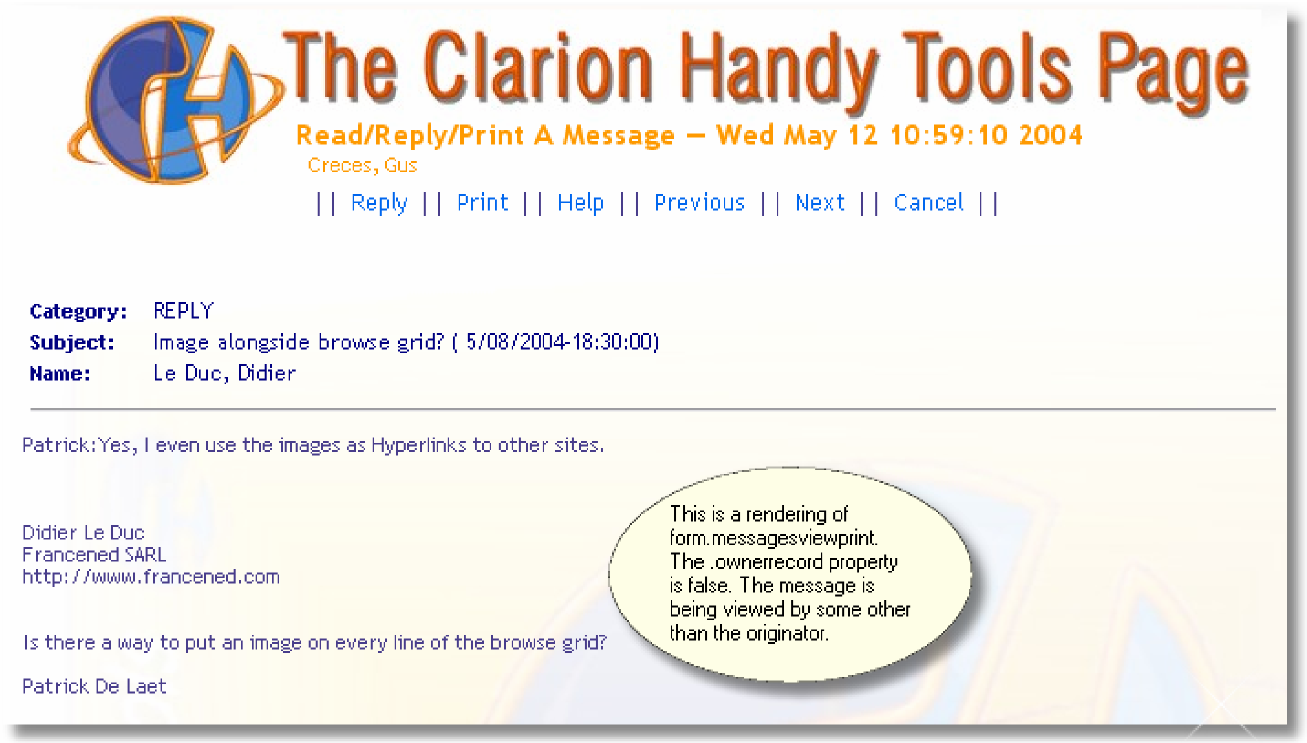
Cheers...

Gus Creces
The Clarion Handy Tools Page
http://www.cwhandy.ca/index.html

This is a rendering of form.messagesviewedit. The .ownerrecord property is true. The message is being created/changed by the message originator.

Illustration 4.4

Messages where the **.ownerrecord** property is false are presented with **form.messagesviewprint**. That is, in non-editable, html-web-page format also illustrated below.


The Clarion Handy Tools Page
Read/Reply/Print A Message – Wed May 12 10:59:10 2004
Creces, Gus
|| Reply || Print || Help || Previous || Next || Cancel ||
This is a rendering of form.messagesviewprint. The .ownerrecord property is false. The message is being viewed by some other than the originator.

Category: REPLY
Subject: Image alongside browse grid? (5/08/2004-18:30:00)
Name: Le Duc, Didier

Patrick:Yes, I even use the images as Hyperlinks to other sites.

Didier Le Duc
Francened SARL
http://www.francened.com

Is there a way to put an image on every line of the browse grid?

Patrick De Laet

Illustration 4.5

Which script name is used for a reply form and which for an edit form is determined in the server on the **BrowserServerHTMLBuilder** template by the developer (see Illustration 4.3). The same script could be requested in all situations. That's up to the developer and is decided at compile-time. However, **what's in the script** and what it does is determined in the script itself and can readily be changed using the provided script editor by the server manager at run-time.

In server scripts see: **(JSTHREAD) Take Thread Button Script.**

The **jsthread.takethreadbutton()** function looks like this:

```
1.  jsthread.takethreadbutton(xobj) {
2.    var nextyear = new Date() ;
3.    var cookiename = document.forms[0].viewid.value.toLowerCase() +
       "lastquery";nextyear.setFullYear(nextyear.getFullYear() + 1)  ;
4.    document.cookie = "" ;
5.    document.cookie = cookiename + "=" + escape(document.forms[0].queryfield.value)
       + "; expires=" + nextyear.toGMTString() ;
6.    document.forms[0].queryfield.value = xobj.threadfilter ;
7.    document.forms[0].editaction.value = 0 ;
8.    document.forms[0].action = "KQY$" ;
9.    return document.forms[0].action;
10. }
```

This script isolates a single message thread, such that only an original message placed via the messages "Insert" menu, and any replies placed via the "Reply" button on the messages form, are displayed in the browse. This is possible because message records designated as "Replies" have the BOD:ID of the parent message (the message to which they are replying) added by server logic to their BOD:ThreadID field.

The parent record has its BOD:ThreadID set to its own BOD:ID. To fetch any thread, then, we only need to access any individual thread member and ask for all records with the same BOD:ThreadID value.

The first set of lines above have to do with storing the current query to a cookie record on the client browser's hard disk. The query being stored describes the browse before the "Thread" button is clicked. Why store this query? If you check out your messages browse you'll see it has a "Recall" button. That button is used to undo or "Recall" the browse to its pre-thread state. Consequently, the user can move into a thread with "Thread" and back out of a thread with "Recall". Each "Recall", does a fresh fetch from the server in case any newly added records matching the original browse query need to be included in the browse.

Code line 6 above assigns **xobj.threadfilter** to the page form's **.queryfield.value**. This is a filter statement, provided by the server as part of the data object for every row, and it recalls all the members of any thread. Sometimes, of course, there is only the parent record in a thread, if it has not yet been replied to. The next line, assigns zero to the page form's **.editaction.value**. An **editaction.value** of zero can be found again in **HNDEQUSK.CLW** to be described as **ACTION:HttpBrowse**, a browse.

The next line, assigns request "**KQY$**" to the page form's **.action** property. From our earlier discussion we know this is **REQUEST:TakeQuery**. And finally, when the function returns the page form is **POST**ed back to the server by the browser.

Anthropomorphically the onclick action of the "Thread" button POSTs a message to the server as follows: "Here, take this query and return the data associated with it to me, packaged for browsing." The server wraps the data objects for the requested records and sends them back with a page that calls for a script called **page.messagesbrowse** to be written out.

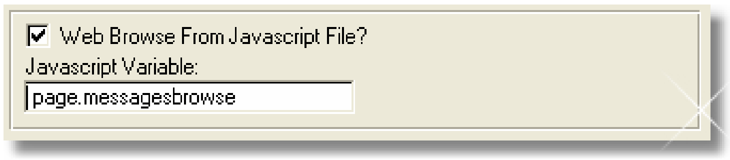

☑ Web Browse From Javascript File?
Javascript Variable:
page.messagesbrowse

Illustration 4.6

Which script name is used for a messages browse is determined in the server on the template named **BrowserServerHTMLBuilder** by the developer, at compile-time (see Illustration 4.6).

What's in the script and what it does is determined in the script itself and can readily be changed in the provided script editor by the server manager at run-time.

ANATOMY OF A BROWSE DATA OBJECT (ARRAY)

It should be obvious by now, from what has been said in other lessons and in this one, that a *browse* data object is really a data array with one array element for every record in the browse. JavaScript data arrays look a lot like Clarion arrays. When we reference **brwmsg[100]**, the array name is "brwmsg", while [100] indicates the hundredth element in the array. Normally the array element pointer is a variable; hence we tend to use expressions like **brwmsg[datarow]**.

Below are two, real-world browse records from a "Thread" button request placed on the CHT support server. Let's give these items the official name **Browse Data Objects**.

Notice that the object names are **NEW**ed as **brwmsg[1]** and **brwmsg[2]**. The second record is the parent. It has a **.bodid** of 11311 and a **.bodngthreadid** of 11311. That gives it away (as being the parent record).

The remaining record is a child of this one because only its **.bodngthreadid** is 11311. Note that these are records formatted for a messages browse. They do not contain the entire message. Only the first few hundred message characters are included followed by "More: (xxx)...". Since messages can be up to 50000 bytes in length it would be pointless and inefficient to send up the entire message string for each browse record. At such time as when we need to see any individual message, we can use the **.fetchfilter** property of that message record to obtain the entire record from the server.

Now look for two other things discussed earlier in this lesson. The field called **.fetchfilter** contains a valid CHT query: **BOD:ID = 11313**. This is the query used by the browse "View/Edit" or "View/Reply" button to bring this record back for edit or reply. To recall a browse record you don't need to know its back-end name or its back-end ID. You only need to know that the necessary query to recall it is stored in its **.fetchfilter** property.

That saves a lot of sweat and guesswork. The field called **.threadfilter** contains a valid CHT query: **BOD:NGThreadID = 11311 ORDER BY -DATE,-TIME**. This is the query applied by the browse "Thread" button. In fact, this is the very same query that spawned these two message records.

Finally, notice that the top record has an **.ownerrecord** value of 1 or True. That indicates the message record was created by the same person who requested this record set.

```
function obj_brwmsg1() {
   this.bodngmemberid = "1596" ;
   this.bodupdated = "1" ;
   this.bodid = "11313" ;
   this.bodngthreadid = "11311" ;
   this.boddatelogged = "10/29/2003" ;
   this.bodtimelogged = "10:32:47" ;
   this.bodmsgsize = "958" ;
   this.bodname = "Creces, Gus" ;
   this.bodcategory = "REPLY" ;
   this.bodsubject = "RE: (10/29/2003- 9:19:38) Web App Participants" ;
   this.fetchfilter = "BOD:ID = 11313" ;
   this.threadfilter = "BOD:NGThreadID = 11311 ORDER BY -DATE,-TIME" ;
   this.ownerrecord = "1" ;
   this.bodmessage = "All you need is a DSL, ADSL or CABLE " +
            "(always on) web connection. " +
            "We provide a test server and all the necessary software. " +
            "The course deals w<br>More: (958)..." ;
}
brwmsg[1] = new obj_brwmsg1();
```

This is message 1 in a browse message array. Its **.ownerrecord** property is one or true indicating that the person browsing is on this browser row seeing his/her own message.

```
function obj_brwmsg2() {
   this.bodngmemberid = "34" ;
   this.bodupdated = "1" ;
   this.bodid = "11311" ;
   this.bodngthreadid = "11311" ;
   this.boddatelogged = "10/29/2003" ;
   this.bodtimelogged = "9:19:38" ;
   this.bodmsgsize = "274" ;
   this.bodname = "Karamarko, Darko" ;
   this.bodcategory = "QUESTION" ;
   this.bodsubject = "Web App Participants" ;
   this.fetchfilter = "BOD:ID = 11311" ;
   this.threadfilter = "BOD:NGThreadID = 11311 ORDER BY -DATE,-TIME" ;
   this.ownerrecord = "0" ;
   this.bodmessage = "I assumed that participants must " +
            "have their own web server "+
            "(dsl and other options)," +
            "CHT Server Script Concepts - Lesson 4 " +
            "so they could do testing through course. "+
            "So I put myself to secon<br>More: (274)..." ;
}
brwmsg[2] = new obj_brwmsg2();
```

This is message 2 in a browse message array. Its **.ownerrecord** property is zero or false indicating that the person browsing did not create this message.

HOW A BROWSE SCRIPT WORKS

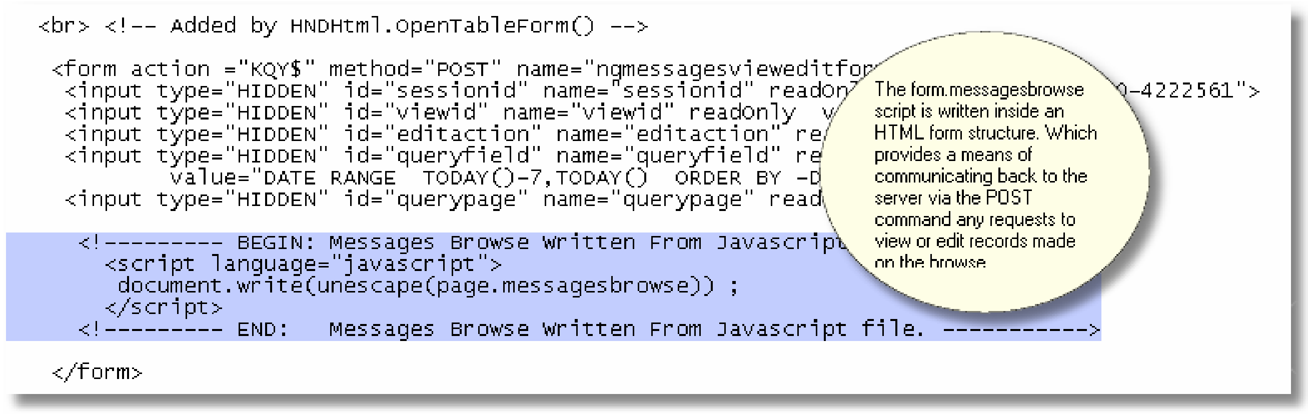
Having examined a browse data array containing **Browse Data Objects**, let's take a look at that portion of the messages browse script that writes this information out to HTML in order to render the browse inside your browser.

The friendly name of the supplied messages browse script is:
(PAGE) 11. Messages Browse Page HTML.

The JavaScript code name for this script is **page.messagesbrowse**.

By right-clicking any messages browse page you can see that server logic is requesting that a script called **page.messagesbrowse** be written out inside the form element of the messages browse page.

```
<!--------- BEGIN: Messages Browse Written From Javascript file. ----------->
 <script language="javascript">
  document.write(unescape(page.messagesbrowse)) ;
 </script>
<!--------- END:   Messages Browse Written From Javascript file. ----------->
```



```
<br> <!-- Added by HNDHtml.OpenTableForm() -->

 <form action ="KQY$" method="POST" name="ngmessagesvieweditfor
  <input type="HIDDEN" id="sessionid" name="sessionid" readon                    0-4222561">
  <input type="HIDDEN" id="viewid" name="viewid" readonly  v
  <input type="HIDDEN" id="editaction" name="editaction" re
  <input type="HIDDEN" id="queryfield" name="queryfield" re
          value="DATE RANGE  TODAY()-7,TODAY()  ORDER BY -D
  <input type="HIDDEN" id="querypage" name="querypage" read

   <!--------- BEGIN: Messages Browse Written From Javascript
     <script language="javascript">
      document.write(unescape(page.messagesbrowse)) ;
     </script>
   <!--------- END:   Messages Browse Written From Javascript file. ----------->

 </form>
```

The form.messagesbrowse script is written inside an HTML form structure. Which provides a means of communicating back to the server via the POST command any requests to view or edit records made on the browse.

Illustration 4.7

In the script example on the next page is the **for** loop portion of the default messages browse script provided with the HNDMTSNG.EXE practice server.

A loop is started which increments a variable called **datarow** from the value in **ndx.msgstart** to the value in **ndx.msgcount**. These two **ndx** (index) variables are provided with every browse data package to tell you how many rows there are in the browse. Variable **ndx.msgstart** normally contains "1" and **ndx.msgcount** normally contains the number of array elements in the browse.

The section commented as: **/* ALTERNATES BROWSE ROW COLORS EVERY SECOND RECORD */** is responsible for alternating the style sheet applied to every other row. This is calculated, using the remainder operator (%) which is the same in JavaScript as it is in Clarion.

Each time the row number, divided by 2, returns a zero (no remainder) the style sheet **bldr_greenbar** is applied to the row. Otherwise the style sheet **bldr_row_container** is applied to the row. Table row **<tr>** information is the first thing written out.

After that the **.boddatelogged**, **.bodtimelogged**, **.bodmsgsize**, **.bodname**, **.bodcategory**, **.bodsubject** and **.bodmessage** fields are written out, each inside a **<td>** or table data tag with its own style sheet to describe how that data element should be formatted. Finally, in the last line, the **</tr>** or table row close tag completes the row.

```
for (var datarow = ndx.msgstart; datarow <= ndx.msgcount; datarow++) {
  /* ALTERNATES BROWSE ROW COLORS EVERY SECOND RECORD */
  if (datarow % 2 == 0) {
    document.write('<tr class="bldr_greenbar">') ;
  } else {
  document.write('<tr class="bldr_row_container">') ;
}

/* INSERT VIEW/EDIT OR VIEW/REPLY DEPENDING IF OWNER RECORD */
```

A section of code omitted here has already been discussed above.

```
  document.write('<td class="bldr_browse_messages_date">' +
   unescape(brwmsg[datarow].boddatelogged) + '</td>');
  document.write('<td class="bldr_browse_messages_date">' +
   unescape(brwmsg[datarow].bodtimelogged) + '</td>');
  document.write('<td class="bldr_browse_messages_size">' +
   unescape(brwmsg[datarow].bodmsgsize) + '</td>');
  document.write('<td class="bldr_browse_messages_name">' +
   unescape(brwmsg[datarow].bodname) + '</td>');
  document.write('<td class="bldr_browse_messages_category">'+
   unescape(brwmsg[datarow].bodcategory) + '</td>');
  document.write('<td class="bldr_browse_messages_subject">' +
   unescape(brwmsg[datarow].bodsubject) + '</td>');
  document.write('<td class="bldr_browse_messages_msg">' +
   unescape(brwmsg[datarow].bodmessage) + '</td>');
```

A section of code omitted here has already been discussed above.

```
  document.write('</tr>') ;
}
```

The complete script of which this code is a component is in server scripts under HTML items. Its name is **(PAGE) 11. Messages Browse Page HTML.**

That's really all that's involved in writing the data portion of the browse. To reformat some of these browse elements, you shouldn't need to touch the browse code at all. Just make a note of the style sheet used by that row item and modify the style sheet characteristics. Below is a code listing for the style sheet called: **bldr_browse_messages_date**.

It's used on the first two fields, **.boddatelogged** and **.bodtimelogged** since they're very similar in size and alignment. The technique of separation of data from the code that displays it and separation of formatting from the code that applies it is very liberating once you understand it. During server building, when you choose the option to display your web data from a JavaScript file the only thing you need to decide at the server level is:

- the JavaScript variable name for the browse building script
- the back-end data variables to be included in the browse

You do not need to decide style sheet names at the server level when the page is being rendered from run-time scripts. That's more flexibly decided at the script-building level.

```
.bldr_browse_messages_date {
  font-family:                tahoma;
  font-size:                  8pt;
  font-stretch:               normal;
  font-style:                 normal;
  font-variant:               normal;
  font-weight:                normal;
  width:                      5%;
  text-align:                 right;
  border-bottom-style:        none;
}
```

In server scripts, under Style Sheet Items, see **(Browse) Messages Date Script.**

To temporarily remove a row element such as the message body from any row, all that's required is to place a JavaScript comment around the **document.write()** statement for that element. For example:

```
/* document.write('<td class="bldr_browse_messages_msg">' +
unescape(brwmsg[datarow].bodmessage)+ '</td>'); */
```

To comment a line of JavaScript code, first highlight the code to be commented, then click the **JavaScript Comment** button in the script editor or pull down the **Format** menu and select the **Comment JS** item as in the screen snapshot below.

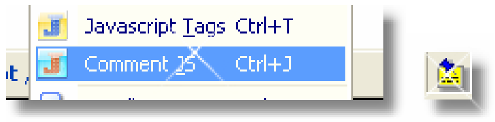
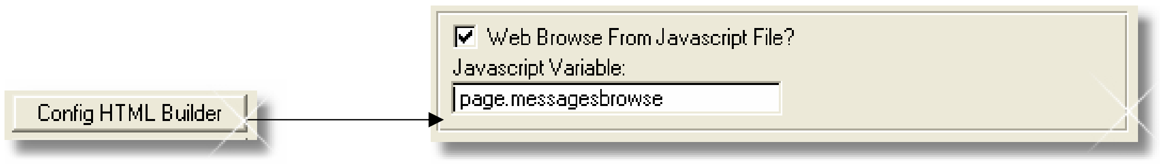


Illustration 4.7

HOW THE SERVER CREATES A BROWSE DATA ARRAY

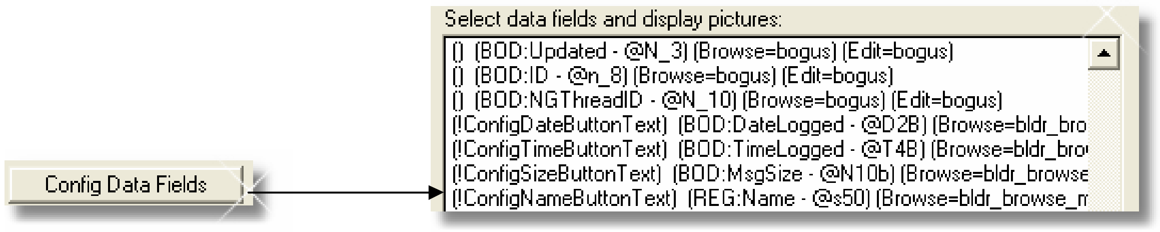
Earlier we showed you that the **BrowserServerHTMLBuilder** template, on the view processing procedure for the messages browse, was configured to produce a web browse via JavaScript file. We told the template about this as follows:



☑ Web Browse From Javascript File?
Javascript Variable: page.messagesbrowse

Illustration 4.8

We set the check box "Web Browse From JavaScript File?" to true, and then told the template which fields we wanted to display as follows:



Select data fields and display pictures:

```
() (BOD:Updated - @N_3) (Browse=bogus) (Edit=bogus)
() (BOD:ID - @n_8) (Browse=bogus) (Edit=bogus)
() (BOD:NGThreadID - @N_10) (Browse=bogus) (Edit=bogus)
(!ConfigDateButtonText) (BOD:DateLogged - @D2B) (Browse=bldr_bro
(!ConfigTimeButtonText) (BOD:TimeLogged - @T4B) (Browse=bldr_bro
(!ConfigSizeButtonText) (BOD:MsgSize - @N10b) (Browse=bldr_browse
(!ConfigNameButtonText) (REG:Name - @s50) (Browse=bldr_browse_m
```

Illustration 4.9

We'll go into more detail on how the fields are configured on the template interface (and from the dictionary) in a later lesson. For the present, suffice to say that by selecting and configuring fields in this manner on the template, we cause the template to write code that calls a method in the CHT Browser Server class **HNDSubscriptionServer (see HNDSUBSV.INC/.CLW)** called **.AddJavascriptExportField()**.

A call is made to this method for each field selected, in order to incorporate that field into the browse data object array. Below is some template generated example server code from the HNDMTSNG.EXE student server that we gave you.

```
xServer.AddJavascriptExportField ('BOD:NGMemberID', BOD:NGMemberID, |
  '@N_8', SIZE(BOD:NGMemberID), Hidden, Read_Only, Disabled, Prompt, 1, '', , )
```

```
xServer.AddJavascriptExportField ('BOD:Updated', BOD:Updated,  |
   '@N_3', SIZE(BOD:Updated), Hidden, Read_Only, Disabled, Prompt, 1, '', , )

xServer.AddJavascriptExportField ('BOD:ID', BOD:ID, '@n_8', SIZE(BOD:ID), |
   Hidden, Read_Only, Disabled, Prompt, 1, '', , )

xServer.AddJavascriptExportField ('BOD:NGThreadID', BOD:NGThreadID, |
   '@N_10', SIZE(BOD:NGThreadID), Hidden, Read_Only, Disabled, Prompt, 1, '', , )
```

The exported fields above are named **.bodngmemberid**, **.bodupdated**, **.bodid**, **.bodngthreadid**. It's been said this several times in earlier lessons but it bears repeating. JavaScript code is case sensitive.

We've introduced the convention of setting all generated variables to lower case so that there is no guesswork when using them in your scripts. JavaScript also cannot tolerate colons (:) in variable names. Hence **BOD:Updated** becomes **.bodupdated**.

The generated Clarion code above, from the HNDMSTNG.EXE server, produces the following JavaScript code packaged as a **Javascript Data Object**.

```
function obj_brwmsg1() {
  this.bodngmemberid = "1596" ;
  this.bodupdated = "1" ;
  this.bodid = "11313" ;
  this.bodngthreadid = "11311" ;
/* SOME FIELDS REMOVED IN THIS EXAMPLE FOR CLARITY */
}
brwmsg[1] = new obj_brwmsg1();
```

ANATOMY OF AN UPDATE FORM DATA OBJECT

From the browse we're able to insert, edit or reply to a message. We've explained already that a browse button calling the **jssubmit.takeedit()** function sends a one-record-query (**.fetchfilter**) to the server with the **.editaction** flag set to **ACTION:HttpEdit**.

Server logic decides whether to apply **form.messagesviewedit** or **form.messagesviewreply** based on whether we've asked to insert a message and/or edit one of our own messages or we've asked to see another person's message and may want to reply to it. Regardless whether we're kicked into edit mode or view mode, the message record returned to us is the same. Only the name of the required script is different. Below is a message record returned from an edit request. We'll call this formally an **Update Form Data Object**.

An **Update Form Data Object** differs from a **Browse Data Object** (array item) in four significant ways:

1) It contains the entire message text.
2) It contains dictionary supplied validation information. For example - the possible choices applicable to a message category field.
3) It contains the dictionary-supplied prompt for that field (You're not obliged to use it in your scripts.)
4) It contains hidden, disabled and read only properties determined by logic that you've built into your server.

The master object name for this update form message record is **bod**. That comes from our dictionary prefix, **BOD**.

All editable fields, **.category**, **.subject** and **.message** are packaged as separate objects inside the **bod** object. They must be addressed with the prefix **bod** followed by a dot followed by the field name followed by another dot followed by a property or method name.

For example: **bod.category.choices**. For clarity, in the JavaScript code I write personally, I tend to distinguish between properties, like **bod.category.value** and functions like **bod.category.validate()** by adding a set of parenthesis.

You don't need to do this but every little bit helps when it comes to understanding your code a few weeks or months after you've written it.

Here are the category field object's components:

bod.category.value	- what the field contains; it has a legal synonym called **bod.category.me**
bod.category.choices	- valid values for this field supplied by the dictionary
bod.category.hidden	- is this field hidden i.e. not displayed on the form at all
bod.category.disabled	- is this field to be displayed disabled
bod.category.readonly	- is this field to be displayed but not-editable
bod.category.prompt	- the dictionary supplied prompt
bod.category.validate()	- this is a function called automatically on the field's onchange event

Notice that the **validate()** function calls a server-supplied generic function called **inlist()**, described below, to ensure that the value inserted into **bod.category.value** is one of the legal values supplied in **bod.category.choices**.

Here are subject field object's components:

bod.subject.value	- what the field contains; it has a legal synonym called **bod.subject.me**
bod.subject.hidden	- is this field hidden i.e. not displayed on the form at all
bod.subject.disabled	- is this field to be displayed disabled
bod.subject.readonly	- is this field to be displayed but not-editable
bod.subject.prompt	- the dictionary supplied prompt
bod.subject.validate()	- this is a function called automatically on the field's onchange event

Notice that the subject field does not include a **choices** property. It will accept almost any value except blank. This field's **validate()** function calls a generic server-supplied validation function called **notblank()**, also described below.

Here are the message field object's components:

bod.message.value	- what the field contains; it has a legal synonym called **bod.message.me**
bod.message.hidden	- is this field hidden i.e. not displayed on the form at all
bod.message.disabled	- is this field to be displayed disabled
bod.message.readonly	- is this field to be displayed but not-editable
bod.message.prompt	- the dictionary supplied prompt
bod.message.validate()	- this is a function called automatically on the field's onchange event to make sure it only contains valid data

Notice that the message field also does not include a **choices** property. It will accept almost any value except blank. This field's **validate()** function calls a generic server-supplied validation function called **notblank()**, also described below.

```
function obj_bod() {
  this.ngmemberid = "1596" ;
  this.updated = "1" ;
  this.id = "11313" ;
  this.ngthreadid = "11311" ;
  this.datelogged = "10/29/2" ;
  this.timelogged = "10:32:4" ;
  this.msgsize = "958" ;
  this.category     = null ;
```

```
      this.initcategory = function() {
         this.me           = "";
         this.value        = unescape("REPLY") ;
         this.choices      = unescape("QUESTION,REPLY,CHAT,SUGGESTION,NEWS,REPORT,CONTACT,")
   ;
         this.hidden       = false ;
         this.disabled     = false ;
         this.readonly     = false ;
         this.prompt       = unescape("Category:") ;
         this.validate     = function() { return inlist(this.me,this.choices); } ;
         this.init         = function() {
           this.me         = document.forms[0].bod_category;
           this.me.onchange = function() { return true; } ;
           this.me.value   = this.value ;
           setchoices(this.me, this.value, this.choices) ;
         }
      }
      this.category = new this.initcategory()

      this.subject     = null ;
      this.initsubject = function() {
         this.me           = "";
         this.value        = unescape("RE: (10/29/2003- 9:19:38) Web App Participants") ;
         this.hidden       = false ;
         this.disabled     = false ;
         this.readonly     = false ;
         this.prompt       = unescape("Subject:") ;
         this.validate     = function() { return notblank(this.me); } ;
         this.init         = function() {
           this.me         = document.forms[0].bod_subject;
           this.me.onchange = function() { return true; } ;
           this.me.value   = this.value ;
         }
      }
      this.subject = new this.initsubject()

      this.message     = null ;
      this.initmessage = function() {
         this.me           = "";
         this.value        = unescape("Darko:%0D%0A" +
         "All you need is a DSL, ADSL or CABLE (always on) web connection.%0D%0A" +
         "We provide a test server and all the necessary software.%0D%0A" +
         "The course deals with building web servers using the CHT tool kit.%0D%0A" +
         "%0D%0A" +
         "Gus Creces%0D%0A" +
         "The Clarion Handy Tools Page%0D%0A" +
         "http://www.cwhandy.ca/index.html%0D%0A" +
         "%0D%0A" +
         /* A PORTION OF THIS MESSAGE HAS BEEN REMOVED FOR BREVITY * /
         "<SNIP>%0D%0A" + "") ;
         this.hidden       = false ;
         this.disabled     = false ;
         this.readonly     = false ;
         this.prompt       = unescape("Message:") ;
         this.validate     = function() { return notblank(this.me); } ;
         this.init         = function() {
           this.me         = document.forms[0].bod_message;
           this.me.onchange = function() { return true; } ;
           this.me.value   = this.value ;
         }
      }
      this.message = new this.initmessage()

   }
   var bod = new obj_bod();
```

The fields not designated "editable" are not packaged as separate objects inside the **bod** object. They're simply properties that can be read but not changed and written back to the data base. This was decided on the **BrowserServerHTMLBuilder** template at data design time.

You've told the server via your dictionary and the CHT templates how you want the information from a given file record to be packaged. This is no different than the desktop applications you build. When you build a browse or a form for a desktop app you display certain fields on the browse/form and not others. It's a design decision.

The same kinds of design decisions that you make on a desktop application need to be made when you build a dedicated data server like the HNDMTSNG.EXE test server we've given you. With CHT web applications you have the extra flexibility of having the user interface entirely separated from the data so that you can totally re-work the look and feel of the app without having to re-visit the Clarion code that manages the back end data (unless you want to).

Please take notice, that there is a significant difference between data packages intended to be used in a browse (Browse Data Objects) and the data packages intended for an update form (Update Form Data Objects). The latter are considerably more intelligent and when based on a well-designed data dictionary, they come with all the necessary validation logic built-in, so that in your page scripts, you do not need to overly concern yourself about data validation. Every data object takes care of its own data validation needs following your dictionary data design.

SOME SERVER-SUPPLIED GENERIC VALIDATION FUNCTIONS

The server includes with every Update Form Data Package a number of dedicated functions that handle things like validation, and web form initialization. Because we discussed a few of these earlier, we'll present some of them here and explain them briefly.

The ones presented here correspond to the data Validity Checks settings assigned in a Clarion data dictionary (see illustrations 4.10). The next lesson, dealing with forms, will cover these and other server supplied functions in greater detail.

The **inrange()** function checks for the correct data range in form data. It works with both alpha and numeric data.

```
function inrange(xfld, xlow, xhigh){
   if ((xval.value >= xlow) && (xfld.value <= xhigh))  {
      return true ;
   } else {
      alert(xfld.id + "  must be in the range of:\n" + xlow + " and " + xhigh) ;
      return false;
   }
}
```

The **inlist()** function ensures that a given field's value can be found in a list of possible choices for that field.

```
function inlist(xfld, xvaluelist){
   var thislist = xvaluelist ;
   if (thislist.indexOf(xfld.value) != -1)  {
      return true ;
   } else {
      alert(xfld.id + "  must contain one of the following values:\n" + thislist) ;
      return false;
   }
}
```

The **isboolean()** function ensures that a given field's value is either true or false.

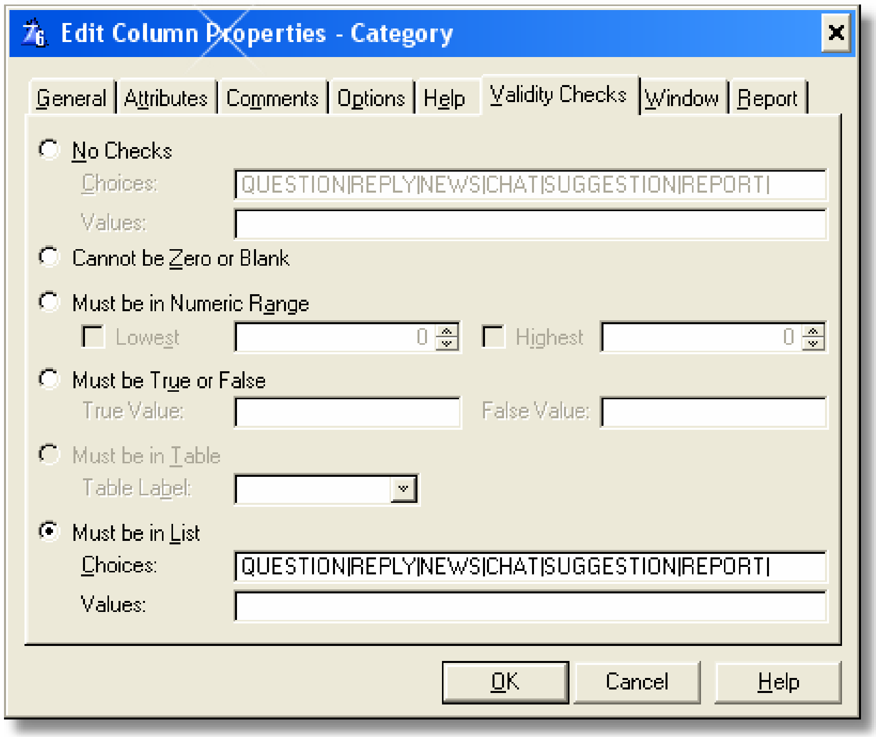
```
function isboolean(xfld, xlow, xhigh){
   if ((xfld.value == 0) || (xfld.value == 1))  {
      return true ;
   } else {
      alert(xfld.id + "  must contain either 0 or 1. \n") ;
      return false;
   }
}
```

The **isblank()** function checks to see whether a field has been left blank.

```
function isblank(xfld){
   for (var i = 0; i < xfld.value.length; i++) {
      var c = xfld.value.charAt(i) ;
      if ((c != ' ') && (c != '\n') && (c != '\t')) return false;
   }
   return true;
}
```

Validation calls to these functions are made automatically with the **onchange** event of editable fields. JavaScript's onchange event is roughly equivalent to Clarion's **accepted** event.

Whether such a validation call is made can be decided in the data dictionary on the Validation Tab or on the **BrowserServerHTMLBuilder** template. A field like message category could be set up as in the screen snapshot below. With the choices hard-coded direct from the dictionary.



- ( ) No Checks
  - Choices: QUESTION|REPLY|NEWS|CHAT|SUGGESTION|REPORT|
  - Values:
- ( ) Cannot be Zero or Blank
- ( ) Must be in Numeric Range
  - [ ] Lowest 0 [ ] Highest 0
- ( ) Must be True or False
  - True Value: False Value:
- ( ) Must be in Table
  - Table Label:
- (•) Must be in List
  - Choices: QUESTION|REPLY|NEWS|CHAT|SUGGESTION|REPORT|
  - Values:

Illustration 4.10

For the HNDMTSNG.EXE server, however, we preferred to give the server manager control over the field choices and have incorporated a server variable (**system.messagecategorychoices**) that allows field choices to be changed at run-time. These are stored in a data base and transferred to the category field's update form data object validation function by the **BrowserServerHTMLBuilde**r template.

LESSON 4 SUMMATION

In this lesson, we've reviewed in detail the answers to investigations we asked you to make in lesson 3.

These involved:
1) Modifying one of the existing scripts to take advantage of a data property supplied with every registrant record to disable a form button
2) Combining two scripts into one and disengaging the script no longer in use
3) Examining the scripts responsible for calling browse update forms and browse threads

Finally, we covered in detail, **Browse Data Object** packages and how JavaScript data arrays coming from the server are used by browse rendering scripts to draw the data grids we call browses. We further examined some of the template settings required to make a **CHT Browser Data Server** generate the Clarion code that creates these Browse Data Objects. Also covered, were Update Form Data Objects with an eye to noticing the considerable difference between data objects targeted for update forms and those targeted for web browses. We examined the auto-validation nature of Update Form Data Objects and how validation settings were communicated upward from the dictionary and templates to the web pages via server logic provided by the **CHT Browser Server Templates**.

UPCOMING...

Next lesson we'll show you the inner workings of web update forms and how they work.

LESSON 4 EXCERCISES

1) The **sig.readonly** property is used in approximately 5 server scripts.
a) Name them and explain how you found this out and how long it took you.
b) Explain how this property is set to true or false.
c) Explain how this property is used in two of the five scripts (be specific)

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.

3) In lesson 3 we had you move the contents of the **title.common** script to the **head.image** 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 already have some code there that you can alter or add to.

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 form code:

```
  <form action ="http://65.95.83.248:443/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,-TIME">
    <input type="HIDDEN" id="currentquery" name="currentquery" readOnly  value="">
    <input type="HIDDEN" id="defaultquery" name="defaultquery" readOnly
                 value="DATE RANGE NOW ORDER BY -DATE,-TIME">
    <!--------- BEGIN: Query Control written from Javascript file. ----------->
    <script language="javascript">
      document.write(unescape(form.messagesquery));
    </script>
   <!--------- END: Query Control written from Javascript file. ----------->
 </form>
```

Talk to you next lesson.

Cheers...
Gus M. Creces
The Clarion Handy Tools Page