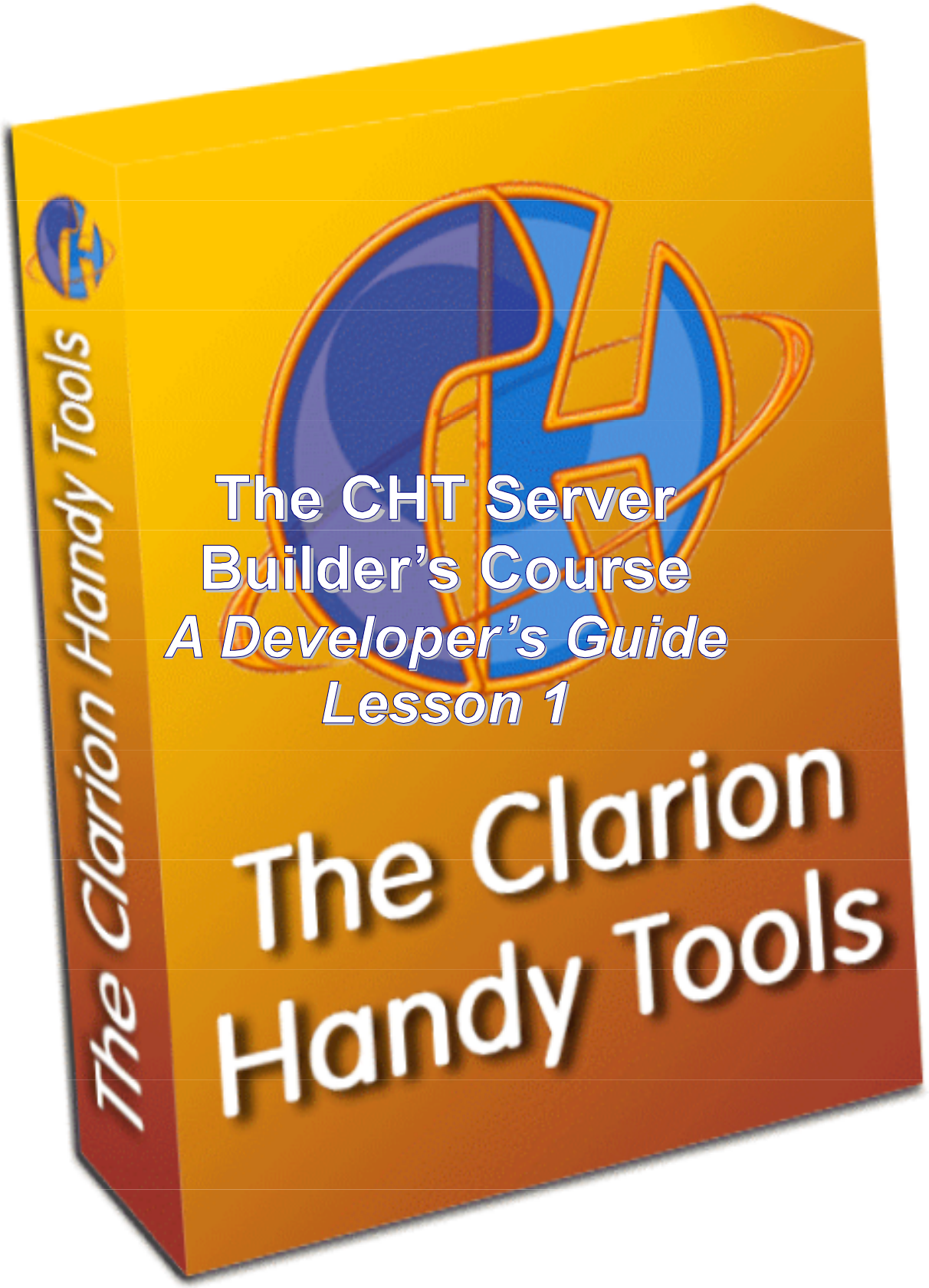
The Clarion Handy Tools
The CHT Server Builder's Course
A Developer's Guide
Lesson 1
The Clarion Handy Tools

PRELIMINARIES

Before you begin reading this first lesson (starting at the **Background** section below), please do the following:

- **Download** a training server called **hndmtsng.exe** from our website.
  The download link is:
  http://www.cwhandy.com/pcdemos/hndmtsngsetup.exe. The password for this install is the same one we gave you when you first became a CHT subscriber (not your serial number). If you can't remember what that was, please send us an email and ask.

  When this server installs, a short essay called "Getting Started With Your CHT Web Group Server" pops up. Please follow the setup instructions provided there exactly**.** As a CHT subscriber you do not need to pay for this fully functional Web Group Server. Your system server credentials as a CHT web app student consist of the single word: **WEBGROUP** in upper case. This value has already been entered for you so there's no need to change it.

  Get your server up and running and try it out from another machine not on your network to make sure that it's working correctly as an INTERNET server, not an INTRANET server. One of the exercises requires you to have this server running so that we can try it out for ourselves from our location. This app will not run in Windows 95, 98 or ME. In fact, one of the later operating systems is recommended, Windows XP or Windows 2000. If you don't have Win NT, WinXP or Win2K to run this on, or if you're still using a slow dialup web connection, as opposed to DSL or Cable, please excuse yourself from this web app course and sign up for a later one when you have these things.

- **Get to know** the server and understand how it works.
  Later in this series of lessons, we'll give you the server's source .APP code so that you can begin to understand both the server side *and* the client side (the browser side) of a web transaction.

  The first five lessons deal primarily with understanding the client side. A web server, serving to a browser, has specific behaviours built into it to accommodate what browsers are able to do (and not do) so it behoves you to understand completely and thoroughly how browsers work. Once those mechanics are understood, the inner workings of this training server - and web servers in general, are more easily explained and understood.

- **Continue** with the remainder of this lesson when you have your server configured and running by following the "Getting Started" outline. The exercise responses required from you should be sent via email to **support@cwhandy.com**, to my attention.

  The exercises and server setup links are due two weeks from now. If you have them done earlier, feel free to send them earlier. But this is not a race and we're all busy with our lives, so take the time necessary to get as familiar as you can with the tools you're being given.

  Ask questions if you must, or if you get stuck. Don't ask questions if you can figure something out for yourself. Feel free to communicate with other individuals taking this web app course.

Here is a complete outline of the entire course as it is currently planned (contents approximate):

1. CHT Server Script Concepts Lesson 1 - CHT Browser Server Review Of Generations 1 and 2
2. CHT Server Script Concepts Lesson 2 - Dynamic Web Pages And JavaScript Data Objects
3. CHT Server Script Concepts Lesson 3 - Anatomy Of a Browser Server Web Page
4. CHT Server Script Concepts Lesson 4 - Browse And Data Form Scripts Using Data Objects
5. CHT Server Script Concepts Lesson 5 - HTML Forms Theory
6. CHT Server Building Concepts - Lesson 1 - Building A Static Page Server
7. CHT Server Building Concepts - Lesson 2 - Building A Static Page Server
8. CHT Server Building Concepts - Lesson 3 - Building A Dynamic Page Server
9. CHT Server Building Concepts - Lesson 4 - Building A Dynamic Page Server
10. CHT Server Building Concepts - Lesson 5 - Building A Dynamic Page Server
11. CHT Server Building Concepts - Lesson 6 - Building A Dynamic Page Server
12. CHT Server Building Concepts - Lesson 7 - Building A Web Server And Client For Live-Update
13. CHT Server Building Concepts - Lesson 8 - Building A Web Server And Client For Live-Update
14. CHT Server Building Concepts - Lesson 9 - Building A Web Server And Client For Live-Update
15. CHT Course Wrap up Plus Show And Tell

Other topics being considered beyond these for the latter part of 2004 and into 2005:
4 or 5 lessons (Building A CHT Internet Client Server And Clarion Client)
3 or 4 lessons (Building an NNTP version of the CHT Web Group Server For Use With O.E.)

Welcome Aboard!
Gus M. Creces
The Clarion Handy Tools Page
support@cwhandy.com

BACKGROUND

The following section is from a white-paper that CHT produced in October of 2001. That document is still available on our website, but we're repeating it here because it continues to accurately reflect the underlying philosophy on which CHT web tools are based.

The Ubiquitous Browser

Every new computing device sold these days, whether Apple, Unix, Linux or Windows based, PALMs, hand-helds, even some cell phones, ship with one common denominator, a more-or-less-universal "data appliance" called a browser that has the potential to supplant the desk top applications you've been in the habit of delivering. When this realization first hit us, we rationalized for quite some time, as you may have done, or are still doing, that the programmatic control you have over a browser in no way approaches the control achievable with a good, solid, desktop application. This was absolutely true then, and is still true today - painfully true, in fact, if you've ever wrestled with conflicting HTML and JavaScript implementations in the browsers out there. You may have rationalized too, as did we, that since browsers were not an adequate replacement for solid applications, they were simply the "flavour or the month" and would go away or be supplanted by something better.

But guess again - browsers are ubiquitous. Their numbers are not going anywhere but up. While internet browsers may be far from perfect, they are a user interface and data delivery system - or appliance, if you will - as nearly universal as you can ask for in an imperfect world. If you think about it, every computer out there today, already has your application/appliance running on the desk top the day it first boots up. If you could only figure out how to "program" that appliance and deliver and receive data in a controllable, interactive way - the way your Clarion applications do. The mind boggles with the potential opportunity - at least mine did!

Clarion in Browser Land

At the same time the mind boggles at the prospect of having to leave the comfort of "Clarion Land" and venture unprepared into "Browser Land" which might more accurately be called "Acronym Land" given the plethora of script languages and data description languages - HTML, DHTML, XHTML, XML, JS, VBS, CSS, ASP, PHP and, of course underneath them, TCP/IP, HTTP, IIS and a hundred more.

Looking past the acronyms, one realizes that below it all lies a very familiar goal - that of getting data in front of users in a controllable, interactive manner. In fact the only thing that has really changed is the appliance that ultimately displays the data and acts as a user interface. What lies below that, the delivery and management mechanism is up for grabs, and as Clarion programmers, we have as good a shot as anybody to get in on that action.

In fact, the browser era represents a unique time, like the paradigm shift from DOS to Windows, in which we can reinvent ourselves as developers. Since Clarion already delivers a large piece of what you need to succeed in Browser Land, we asked ourselves how, as tool makers, CHT could help to give Clarion developers the same leg up in browser-application development that they've always had in desk top-application development.

The early results of those thought processes and more than a year of experimenting on our part are here for you to try in our O7A-1 build *(Remember as you read this that it was written in October 2001)*. What you're seeing and getting in this build is only the tip of the iceberg. The theory underlying our HTTP initiative is familiar and comfortable for all Clarion developers. You apply a template, fill in the prompts, and a clarion class is latched into your application. The class writes HTML, XHTML and JavaScript as required (based on the template prompts) to build a browser-based interface to your data with TCP/IP connectivity wrapped in as needed.

How Is This Any Different?

How is this CHT offering different than the good things you already have at your disposal, CWIC and ClarioNet? We have kept foremost in our minds the need to differentiate - no point duplicating what has already been done - and the need to carefully target what we feel is rapidly becoming the world's most familiar data interface appliance - the browser. *(Since this was written there are two new entries in this area: Soft Velocity's ASP Templates and an ASP generation technology called Phoenix from a Dutch company.)*

Difference #1 - The Connection

Your CHT Browser Server application has a direct TCP/IP connection to the browser itself. It does not require an internet server front end like IIS or Apache, nor does it require that Clarion's Application Broker be installed. This reduces the preparatory overhead drastically and virtually eliminates a host of potential ills and worm holes that result from badly configured internet servers. Any system with a TCP/IP address can act as a data server to any other machine that has a browser and the required login password. You are 100% in control of what types of requests your CHT Browser Server will honour and what it won't, and who you let in or who you send packing.

Difference #2 - The Data Appliance

ClarioNet is essentially an every man's "thin-client" computing tool targeted to compete in the same arena as CITRIX and the like. A binary interface appliance (i.e. a program) is required at the client side to reproduce the application window. While CWIC targets the browser, it is also built on an "application window" paradigm. It attempts to reconstruct the original application window, and its event model inside the target browser. A CHT Browser Server application makes no attempt to act like or look like a standard desk top application inside a browser. It acts, looks and feels like any HTML page. The components of the HTML pages you create - buttons, entry fields, links, etc. - are generated for you by templates much like the control and extension templates that have

always been available to help you build desk top application windows. Unlike either of the aforementioned products, and more like SQL, a CHT Browser Server application displays data only after the user decides what it is they want to see. From the browser window, users formulate queries, describe how the data should be ordered, and press "Send" or "Submit". A complete result set, based on the query, is sent back to the browser packaged in generic HTML so as to be readily compatible with a variety of browsers running on any hardware or operating system. There is no further overhead to scroll data up and down one page to the next. The scroll bar on the browser can already do that, free of charge, no code required. If the data size gets out of hand, users learn very quickly that with a query, you "get what you ask for".

Difference #3 - The Back End

A CHT Browser Server application does not become "all internet" or "all desk top" by flipping a template switch. The internet component of your application is not a desk top application masquerading as an internet application. Its purpose is to serve your data to a browser in a form that browsers most readily understand - generic HTML and JavaScript. The demo application we've provided with this build (HNDDTASV.APP) happens to be 100% internet application. It has only two major components: a server/connectivity component built on a standard ABC window template and a data packaging component built on a standard ABC process.

Evolution, Not Revolution

Rome wasn't built in a day. Neither were The Clarion Handy Tools. The Browser Data Server templates and classes provided with this build, while fully functional, are far from complete. You may not be aware that HTTP internet connectivity has been lurking in your tool set for more than a year. We've been honing and perfecting it in the background. If you weren't aware that it was there, no harm done. In the Clarion environment it's the templates that ultimately bring all low-level functionality within reach of the developer. So when we say "far from complete" we mean there are plenty of templates and OOP Classes yet to come on the way to fulfilling this particular vision.

As this HTTP initiative evolves we'll be seeking and expecting your feedback. If there's one thing we've learned, no matter how "cleverly" we think we've abstracted a problem/solution pair, there's always another spin on a problem that we may not have considered.

Have fun with this. Start with the HNDDTASV.APP application that we've built for you and get to know and understand it, both as an executable, and as source code. If you're not yet a CHT subscriber and you want to get in on the action, you're welcome to join for the price of a subscription.

Gus M. Creces
The Clarion Handy Tools
October, 2001
support@cwhandy.com

CHT BROWSER SERVER GENERATIONS ONE AND TWO

The description that you just read in this two and a half year old white paper, describes surprisingly accurately what transpired with CHT web tools over the following 18 months in *Generation One* and *Generation Two* of The Clarion Handy Tools *Browser Server* technology.

The essence of that work was server technology that generated HTML-wrapped data returned from a query and provided data interactivity. The following chart outlines both *browser-side* and *server-side* what happens when web-app pages are requested and displayed. This chart is by no means comprehensive since there are a lot of other permutations, even in a relatively uncomplicated application like the Web Group server.

	BROWSER	**SERVER**
1	A browser user contacts your CHT Browser Server application by entering a URL into the browser command line: http://www.yourserver.com	The server responds by sending an introductory page with some menu options.
2	The browser user decides to log in. He clicks the "Login" menu.	The server sends a page that requests login credentials. What sort of credentials are required is determined by the server designer (developer) based on the "Subscriber" data base he decides to attach to his server application.
3	The browser user enters his login credentials, which the server designer has decided should be last name, email address and a server generated login number, followed by a click on the "Send" or "Login" button provided.	The server checks the provided credentials against its data base and branches accordingly. If the credentials are wrong, it sends a page that says so and explains alternatives. If the credentials are correct it sends a page and a set of menus that provide access into the data bases. Each database, or data view, has a separate entry point fronted by a query control in which the browser user must describe the data they wish to see. From this point forward every page sent to this individual, and every post this individual sends back to the server is identified by a unique, one-time-only "tracking" number that's been attached to this relationship. That tracking number has no real meaning outside this relationship. Once the relationship has ended (via logout), any given tracking number is obsolete. Tracking numbers are never repeated and cannot be reused.

	BROWSER	SERVER
4	The browser user receives a page with some explanatory text, a "query" control and several menu items, some of which lead to other "query" controls. Since this is a "Message Server", he/she navigates, via the menu to the "Messages" query control and enters a query that describes which messages he/she wants to see, for example: DATE RANGE THISWEEK, followed by a click on the "Send" or "Query" button.	The user's query is posted back to the server along with some embedded information identifying the back-end view to which the query pertains. Server logic passes the query string to a Clarion ABC process that is able to resolve the query in the correct data base or data bases. That is, it is able to find the records that match the query. The same Clarion process has been programmed, via CHT template to "wrap" the data in HTML in such a way as to simulate a sorted browse and gives the "wrapped" data back to the server's send-receive componentry. Next, the server encloses the HTML-wrapped data with an HTML page header, optional footer and some navigation menus and sends the whole package back to the requesting browser. Since this is a multi-tasking server, other requests may have come in while it was doing all this, but it knows to which specific browser this data package should be sent because of the tracking number. It returns the data in the background on a true Window thread while it goes on with servicing other browser requests.
5	The browser user sees a page resembling a browse that displays the "Messages" data he/she requested. Since there were quite a number of messages in the past week he decides to narrow his search somewhat and enters a new, narrower query. He navigates the menus back to the "Messages" query control and enters: DATE RANGE NOW, followed by a click on the "Send" or "Query" button.	The server intercepts the post, sent to it by the browser, and identifies the user who sent it, via tracking number. It isolates the query string and the back end view against which this query must be processed. The query is branched to the correct Clarion process, where matching data is HTML wrapped as before. This is then passed back to the server's "final-packaging" routines which expedite the package back to the browser in the form of a new HTML page.

	BROWSER	SERVER
6	The page transmitted by the server is intercepted by the browser and a new data "browse" is presented. This time the information displayed represents only messages posted that are stamped with today's date. (DATE RANGE NOW). Along the right-hand side of the browse, two per-record, is a row of buttons marked "Open" and "Thread". The browser user wants to read the entire body of a message so he clicks it's Open button.	Underneath the Open button that the user clicked, the server embedded some logic that caused the browser to post a request back to the server with a query that isolates a specific message using the message's unique ID. Along with this comes an action flag that indicates how the returned data should be wrapped, not in a "Browse" package this time, but in a "View" package. A view package presents a single record for reading, along with a number of controls that allow for both canceling the view and replying to it. This query is again processed by the very same ABC process that handles browse building.
7	The browser user now sees a message presented as a standard, scrollable web page. Not edit capabilities are offered for this message, since a message may only be edited and deleted by the person who created it. However, a REPLY menu is provided which allows the message to be replied to. He/she clicks the REPY button.	Underneath the reply button that the user just clicked, the server embedded some logic that caused the browser to post a request back to the server, again isolating the same specific message. Along with this came a new action flag that caused the server to repackage the message yet again, not in a "View" package this time but in a "Reply" package. This query with its new action flag is once again handled by the same ABC process that handles browse building and view package building. The process formats the message in reply format (with a salutation and a <SNIP> tag) and actually writes it to disk as a new data base record. This reply may or may not be "saved" by the browser user, so a critical "saved" flag field in this data base record is set to zero. Only if the browser user now enters his/her own message and clicks save, is this record considered saved. If the user were to not complete his reply, and click "Close" instead of "Save" the data base record would never have its saved flag updated to non-zero. The server has a garbage collection routine that deletes any messages older than a configured value that have a zero in saved flag field.

	BROWSER	SERVER
8	The browser user now sees a scrollable, editable HTML edit box called a TEXTAREA. This time it is not set read only since the user will be typing his own message into this text box. At the top of the edit box is the first name of the person to whom the message is addressed, followed by a colon, a couple of blank lines and a <SNIP> tag after which the original message appears. The browser user types his/her message and decides to post it back to the server by clicking the "Save" button provided.	Underneath the save button that the user just clicked the server embedded some logic that caused the browser to recognize this as a "Save" operation. Using the query that describes this message alone, (ID = xxx) the message record is retrieved from the data base and updated with the latest information entered by the browser user. During a save operation the message record's "saved" flag field is updated by 1. This flags the server that this message should be considered viewable (by other browser users) and that it should not be considered for garbage collection. After a save, operation the server recognizes that the user who posted this save operation should be given back the web "browse" from which the view -> reply -> save operation was first initiated. To that end, the server looks in it's on-line members cache using the member's unique tracking number, finds the last browse-initiating query that the member posted and sends the member a web browse based on that query.
9	The browser user now sees the same (or similar) browse from which he originally began his/her view->reply->save sequence. Depending on the scope of the original query that spawned this browse, the user sees their newly posted message now added to the browse. If other members are also posting simultaneously, those messages may appear as well.	An HTTP server can (broadly defined) do only two things. It can intercept requests sent to it and send back responses. It does not normally stay connected to a client browser longer than the time it takes to receive a request and send back a response. The relationship between a browser client and an HTTP server is an "action-reaction" relationship. The browser starts by requesting some action to be taken and the server reacts to this as it has been programmed to do. Aside from following the preset rules that describe how an HTTP message should be formatted (so that the browser knows what to do with it) an HTTP server can pretty much set it's own rules about how requests should be formatted. CHT data servers make no attempt to emulate the behavior of generic web servers such as IIS (Microsoft Internet Information Server).

RECOGNIZE THE PATTERNS

Those are, obviously, not all the combinations of browser/server interaction possible. By taking you through at least some of the steps outlined, you may have begun to recognize a pattern in the interactions described. Since some of these interactions are not so obvious, here are the key ones:

The CHT server is stateless.

This means it does not keep track of what a client browser is doing at any point in time. It keeps track of client browser users with valid accounts who have logged in within the last hour via a tracking number, but it doesn't really have a clue what the clients are up to. The term *stateless* refers to the fact that the server is not responsible for keeping track of the state of windows or conditions in the browser client. Or that you (the browser client user) have scrolled half way down the page and are viewing a message from, say John

Smith. For all it knows, (or cares) you've already closed your browser session, having forgotten to click the "Quit" menu, and are sending an email to a friend. The only thing the server is responsible for doing is listening for requests, rejecting requests from invalid sources and responding to requests from valid sources. The requests it receives, even from valid users must be formatted in a prearranged manner and the responses it sends back are conditional to the request and are also formatted in a prearranged form.

This is how a CHT web data server differs from methodologies like CWIC and ClarionNet. CHT servers are *stateless* and as a consequence a single server can serve hundreds of client browsers. With ClarionNet, the state of the client appliance is controlled by the server. Consequently, there is a one-to-one relationship between client appliances and server applications. CWIC does the same thing, if there are 100 clients connected to a server computer, there are 100 instances of your server application running.

During any single session, the server and browser are more often disconnected than connected.

That is to say in a one-hour session during which a single user reads ten messages and posts three or four of his own, the user's browser and the CHT server servicing it were probably not really "connected" for more than thirty seconds, probably less, but they may have connected and disconnected a thousand times (at the speed of electrons, of course). The whole premise of HTTP communications is to minimize connection time. Each time a new request goes out from the browser a new "TCP/IP Socket" connection is established. The socket is kept open until the server response comes back - or times out - and then it's promptly closed again.

Consequently, any single socket communication contains one transaction loop consisting of a request and a response, after which the socket connection is closed. Here is an anthropomorphic description of what happens during the course of one of these socket connection/transaction loops. The client browser finds your server on the internet or intranet via it's IP address - remember, URLS like www.cwhandy.com are resolved to IP addresses by other web servers called DNS, or Dynamic Name Servers. It pings the server, "Are you there?" which responds with the TCP/IP equivalent of "Go Ahead I'm Listening".

On receiving this acknowledgement, the browser sends its request, perhaps something as simple as http://www.cwhandy.ca/GHP$ (Get Home Page). The server resolves this request and honors it if it can, or responds with the electronic equivalent of "Huh?" if it can't. If an internet worm aimed at a Microsoft server were to make a request of a CHT server it would either respond with "Huh?" or if it had been taught about this worm - which is possible to do - it would respond by closing the connection and saying nothing.

As a consequence of this *mostly disconnected* relationship between a client browser and a CHT HTTP server, during the course of an on-going web session, the HTTP protocol is extremely robust in situations where the connection is less than optimum.

The server "wrapped" all back end data in HTML before returning it to the browser.

Right out of the box, without plug-ins and add-in doodads, most browsers are designed to know at least two languages, HTML and JavaScript (Cascading Style Sheets are an extension of HTML and are therefore encompassed by that term). Java code execution is possible in a browser but not native, out of the box without some extensions and add-in libraries.

Please don't confuse Java with JavaScript. Despite the similarity of the names, they're two different things. HTML (Hyper Text Markup Language) isn't a computer language, per se. It's more of a display description system since it contains no flow control mechanisms like IF, CASE or LOOP. CSS (Cascading Style Sheets) are a means, in HTML, of defining in great detail how something, say a button, looks and assigning a name to that definition so that one can re-use it by invoking its name rather than having to repeat all the little details.

Without CSS, HTML can become the worst kind of wordy, repetitive, unreadable, spaghetti code you could possible imagine. JavaScript is a real computer language that a browser can execute.

Your web browser is, in fact, a run-time JavaScript interpreter. *Remember that*. You'll need to internalize that bit of information because it's key in CHT *Generation Three* web applications. JavaScript can also create HTML on the fly. That is to say, a developer can write some JavaScript code that draws a web screen using HTML that dynamically includes data base information stored in JavaScript variables sent up from the server.

But I'm getting ahead of myself.

The salient point here is that CHT *Generation One* and *Generation Two* CHT servers primarily did the data "wrapping" at the server end based on Clarion compile-time code that you added to your server. A lot like a Clarion desk-top application if you configured your browse to look a certain way on our HTML builder template, that's the way it would come out in the browser.

Execution logic, server commands and browser state information are embedded in the HTML page.

Because a browser-based web application via HTML is stateless, you cannot, (should not) involve the server in having to track client state information. To achieve this, every web page stores a certain amount of "state" information about itself which travels back to the server when a submission or request comes from the browser to the server. The *tracking number* is an example of this. The *action flag*, which determines browse view or reply mode, is another example.

The *view identifier* is another example. It tells the server to which back-end HTMLBuilder process a browser request should be routed for processing.

The *current query* is yet another example of *state* information. It describes the record or records impacted by the *action* taking place. State information normally resides in HTML *hidden* variables. Hidden doesn't mean the user can't see them if he wants to. Primarily it means he/she can't edit or change them. They are not displayed on the web window in any kind of edit control.

Besides state information, all pages that require users to take some action include a FORM section. An HTML *form* is the means by which a web page makes submissions to a server. Every form has an ACTION (a web command) associated with it. The command might be TAKE QUERY (KQY$) or GET HOME PAGE (GHP$) or any one of 60 others.

If you want to see a list of these, open CHT source module HNDEQUSK.CLW and search for REQUEST:, the list is there in the form of programmer-friendly equates. These web commands are purely, a CHT web server concept, used with Microsoft's IIS, they're meaningless.

If you want to see the how CHT code processes these web commands inside the server, open a source module called HNDSUBSV.CLW and search for a method called:
HNDSubscriptionServer.ParseCommand.

You'll see a long CASE statement, each case element dealing with a different REQUEST.

LESSON 1 - SUMMATION

In this first lesson, we provided some background on the philosophical underpinnings of CHT Browser Server technology and how it evolved in the last couple of years. All of the things discussed are still relevant in *Generation Three* (O8A1.0 Build, forward).

In the next lesson we'll point out the chief difference between *Generation Three* and the prior two. **That difference is the reason for the SERVER SCRIPTS component of the HNDMTSNG.EXE application.**

LESSON 1 - EXERCISES

1) What is a dynamic web page and how does it differ from a static web page?

2) Describe briefly, two basic techniques for dynamic web-page creation. (Hint: Server end - Client end).

3) Sign into your own, running newsgroup server and click the login menu. No need to log in. Right click the page and select "View Source". Figure out how to open these files in and editor and deduce what purpose each serves. Describe briefly what each of these lines does.

3a) JavaScript file request hndmtsngcfg.js

3b) JavaScript file request hndmtsngsub.js

3c) JavaScript file request 7XXXX_XXXXXXX_XXXXXXXXXXX.js (names will differ)

3d) JavaScript file request hndmtsngsvr.js

3e) Style sheet request hndmtsngcsty.css

4) Describe briefly how to clear your browser's web cache.

5) Why is clearing the web cache sometimes important (how does that relate to the CHT newsgroup server?)

6) Explain how you go about finding out where your web cache is located.

7) Explain how to configure I.E. to optimally interact with dynamic-page websites like the CHT newsgroup server.

8) Briefly, describe what each of the following does on a web page and what the acronym stands for (if any).

8a) HTML

8b) CSS

8c) JavaScript

9) Modify the introductory text on your server's home page to reflect the theme of your particular NG website using the built in "Server Scripts" editor.

10) Send the URL:PORT or IP:PORT address to me along with a time when I can see the server running.

11) I expect to be able to register myself as a member and get back a registration email.

Gus M. Creces
The Clarion Handy Tools Page
support@cwhandy.com