DYALOG

The tool of thought for software solutions

HTMLRenderer User Guide

Dyalog APL Version 17.1

Dyalog Limited

Minchens Court, Minchens Lane
Bramley, Hampshire
RG26 5BH
United Kingdom

tel: +44 1256 830030

fax: +44 1256 830031
email: support@dyalog.com
http://www.dyalog.com

Dyalog Version 17.1

Revision: 2020010801_171

Contents

1 Introduction

Introduced with Dyalog 16.0, HTMLRenderer is an object which provides a cross-platform mechanism for producing Graphical User Interfaces (GUI), based on Hypertext Markup Language (HTML). As of Dyalog version 17.1 HTMLRenderer is available on Microsoft Windows, Apple macOS, and Linux (excluding the Raspberry Pi). Using HTMLRenderer, your application can use the same user interface code and work in the same way across platforms.

HTMLRenderer is a built-in class, instances of which are created and managed using the Dyalog GUI framework functions `⎕WC`/`⎕WS`/`⎕WG`/`⎕NEW` and `⎕DQ`/`⎕NQ`. User interfaces are defined using HTML, which can, in turn, make references to code and data in a number of additional formats such as JavaScript to manage highly interactive content, Cascading Style Sheets (CSS) for both simple and sophisticated styling, SVG, JPG or BMP for images.

On all platforms, the creation of an HTMLRenderer object causes APL to open a new window and run a copy of the Chromium Embedded Framework (CEF), passing the HTML to the CEF for rendering.

Dyalog APL version 17.1 introduces several new features for HTMLRenderer.

- Support for websockets, allowing asynchronous, bi-directional communication between the APL session and the CEF client window.
- A **`DoPopup`** event that is triggered when the CEF client issues a request for a new window.
- A **`SelectCertificate`** event that is triggered when the CEF client issues a request for a resource that requires a certificate.
- A **`ShowDevTools`** method that will toggle the visibility of the Chromium Developer Tools to inspect and debug from the CEF client.
- Support for several `⎕WC` properties including `Caption`, `SysMenu`, `MinButton`, `MaxButton`, `Sizeable` and `Moveable`. Some properties may not be available on a particular platform because that platform does not have underlying support for the property; setting such a property will have no effect, nor will it cause an error.

The HTMLRenderer can be disabled by setting the ENABLE_CEF environment variable to 0; if ENABLE_CEF is not set or is set to 1 then the HTMLRenderer is enabled (the default).

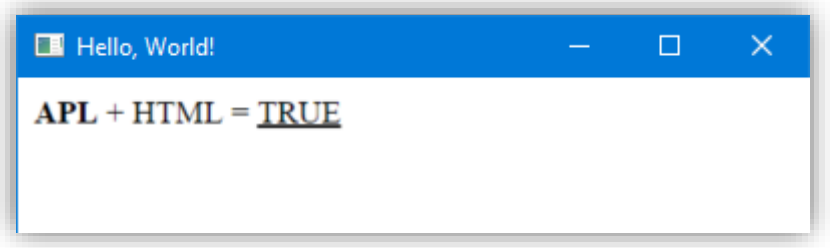
On some platforms the HTMLRenderer and RConnect (the Dyalog R interface) both expect to run in the main thread; attempting to run both in the same process will lead to APL terminating uncleanly; if you want to use RConnect set ENABLE_CEF=0. Attempting to create an instance of HTMLRenderer when ENABLE_CEF is 0 will cause a "`LIMIT ERROR: The object could not be created`" error to be signalled. See section 4, *Technical Overview* for more information on enabling the HTMLRenderer on various platforms.

1.1 Hello World

A minimal example of using HTMLRenderer would be the following: The first line of code below creates an HTML header which includes a title tag – this will set the caption for the form that contains the HTMLRenderer. The second line defines the HTML body, and uses simple HTML tags to mark text up as bold, italic and underlined. Finally, an HTMLRenderer is created, using the header and body as the HTML and setting the size property as well:

```
head←'<head><title>Hello, World!</title><head>'
body←'<body><b>APL</b> + </i>HTML</i> = <u>TRUE</u></body>'
'hr' ⎕WC 'HTMLRenderer' ('HTML' (head,body)) ('Size' (10 20))
```

Under Microsoft Windows, the result will be:



1.2 Other Resources

All HTML applications are based on an initial HTML document. Most modern user interfaces will reference other documents, such as JavaScript and CSS files which contain code that can influence the way the base HTML is rendered, image files, and of course hyperlinks to other pages.

If the HTML contains references to other resources, the CEF will retrieve each one by making an HTTP request. Each request with a URL that matches a triggering pattern in `InterceptedURLs` will generate an `HTTPRequest` event on the instance of HTMLRenderer, which can be directed to a callback function in APL. Requests with URLs that do not match a pattern in `InterceptedURLs` or that match a pattern with a 0 in the second column will cause the CEF to push the request out to the network and see whether an external server is able to service it. `InterceptedURLs` allows an APL application to decide how which content it wants to provide, and to what extent it wants to act as a portal for other services that will provide the rest of the data.

1.3 User Events

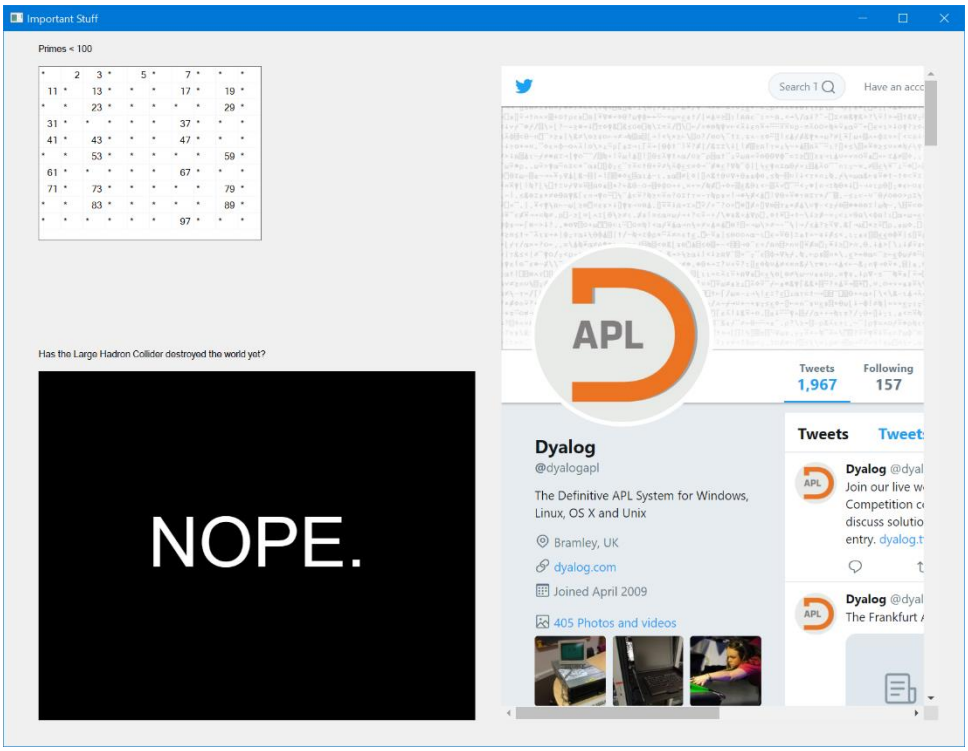
When a user submits an HTML form for processing, or a user interface component which is being managed by JavaScript code wishes to make a server request, this is also done by making an HTTP Request. These requests will also be directed through the same `InterceptedURLs` mechanism. This makes it possible to develop interactive applications where your APL code is responding to user input, as well as providing the content of resources needed to render the UI.

1.4 An Example of a Portal

The following code illustrates how HTMLRenderer objects can be used as children of normal ⎕WC forms under Microsoft Windows. By setting the AsChild property of an HTMLRenderer object to 1, we request that the window be embedded as a sub-form of another window.

```
'f1'⎕WC'Form' 'Important Stuff' ('Coord' 'ScaledPixel')('Size' 820 1100)
)copy dfns pco
'f1.label1' ⎕WC 'Label' 'Primes < 100' (10 40)
'f1.primes' ⎕WC 'Grid' ('*' @ (0∘pco) 10 10⍴⍳100) ('Posn' 40 40)
f1.primes.(TitleHeight TitleWidth CellWidths Size)←0 0 25 (200 255)
'f1.label2' ⎕WC 'Label' 'Has the Large Hadron Collider destroyed the world yet?' (360 40)
'f1.areWeStillHere' ⎕WC 'HTMLRenderer' ('AsChild' 1) ('Posn' 390 40)('Size' 400 500)
f1.areWeStillHere.URL←'http://hasthelargehadroncolliderdestroyedtheworldyet.com'
twitter←'<a class="twittertimeline" href="https://twitter.com/dyalogapl">'
twitter,←'Tweets by dyalogapl</a>'
twitter,←'<script async src="//platform.twitter.com/widgets.js" charset="utf-8"></script>'
'f1.twitter' ⎕WC 'HTMLRenderer' ('AsChild' 1) ('Posn' 40 570)('Size' 750 500)
f1.twitter.HTML←twitter
```

The result after clicking on the "Tweets by dyalogapl" link can be seen below; a form that contains a Windows grid showing prime numbers between 1 and 100 as well as provides live feeds from two external sites. Note that no callbacks have been assigned and no URLs are indicated to be intercepted (`InterceptedURLs` was not set); in this case the HTMLRenderer always goes to the network to satisfy requests for data.



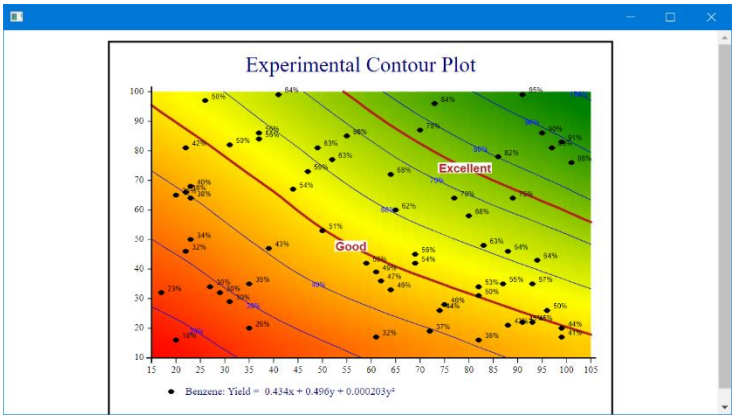
2 Simple Examples

2.1 Render a SharpPlot chart

```
      )LOAD sharpplot
saved…

      'HR' ⎕WC 'HTMLRenderer'

      HR.HTML←#.Samples.Contour.RenderSvg #.SvgMode.FixedAspect
```



2.2 An application with 2 Pages

The function on the next page creates a very simple application with 2 pages: A home page called main and another page called clicked which is displayed if the user follows a link. Initialise the application by calling myapp with an empty right argument; this will cause it to create a namespace containing all the resources, and then create an HTMLRenderer and set the URL property so that it navigates to the first page – and itself as the callback function.

If called with a non-empty argument, the function handles callbacks. It extracts the page name from the URL, which corresponds to a variable in the namespace and returns the value of that variable as the response to the request.

```
     ∇ r←myapp args;root;evt;int;url;size;coord;obj;op;sc;st;mime;hdr;data;meth;page
[1]   ⍝ Serve up a small 2 page application
[2]
[3]    root←'http://myapp/'  ⍝ set the root, requests from CEF will start with this
[4]
[5]    :If 0∊⍴args ⍝ empty args means we're doing Setup
[6]    ⍝ define the "app" in MyApp, 2 static HTML pages
[7]        #.MyApp←⎕NS''
[8]    ⍝ HTML for the "main" page
[9]        #.MyApp.main←'Hello APLers<br/>Click <a href="clicked">here</a>!'
[10]   ⍝ HTML for the "clicked" page
[11]       #.MyApp.clicked←'Thank you!<br/>Click <a href="main">here</a> to go back!'
[12]   ⍝ we want to intercepts all requests that begin with the root
[13]       int←'InterceptedURLs'(1 2⍴(root,'*')1)
[14]   ⍝ whenever we get a request for a resource, call myapp (this function)
[15]       evt←'Event' 'HTTPRequest' 'myapp'
[16]   ⍝ set the initial URL to the "main" page
[17]       url←'URL'(root,'main')
[18]   ⍝ set some window parameters
[19]       size←'Size'(150 300) ⋄ coord←'Coord' 'ScaledPixel'
[20]   ⍝ and off we go...
[21]       'hr'⎕WC'HTMLRenderer' url evt int size coord
[22]
[23]   :Else ⍝ handle the HTTPRequest event
[24]       (obj evt op int sc st mime url hdr data meth)←11↑args
[25]   ⍝ extract the page name
[26]       page←(≢root)↓url
[27]   ⍝ does the page exist?
[28]       :If 2=#.MyApp.⎕NC page
[29]   ⍝ set the HTTP status and text for a successful request
[30]           (sc st)←200 'OK'
[31]   ⍝ set the response data to the new page's HTML
[32]           data←#.MyApp⍎page
[33]       :Else
[34]   ⍝ set the HTTP status and text for a failed (not found)
[35]           (sc st)←404 'Not Found'
[36]           data←'<h2>Page not found!</h2>'
[37]       :EndIf
[38]   ⍝ set the MIME type for the response
[39]       mime←'text/html'
[40]   ⍝ indicate that we've intercepted and handled this request
[41]       int←1
[42]       r←obj evt op int sc st mime url hdr data
[43]   :EndIf
     ∇
```

2.3 A Form with a Button

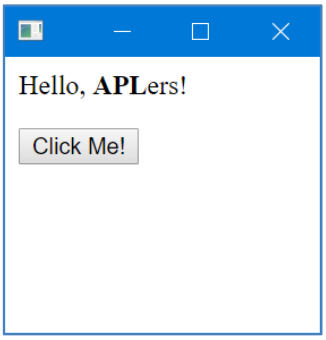
Define a callback function:

```
     ∇ r←my_callback args;obj;evt;op;sc;st;mime;url;hdr;data;int;meth
[1]   ⍝ Our first HTTPRequest callback function
[2]    (obj evt op int sc st mime url hdr data meth)←11↑args
[3]    int←1                              ⍝ Intercept this call
[4]    (sc st mime)←200 'OK' 'text/html'  ⍝ HTTP success code
[5]    url←hdr←''                         ⍝ no url or header
[6]    data←'<h2>Thank you!<h2>'          ⍝ response Data
[7]    r←(obj evt op int sc st mime url hdr data)
     ∇
```

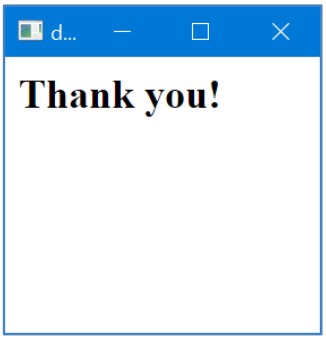
Now, define a form and set up the callback:

```
 'hr' ⎕WC 'HTMLRenderer' '<p>Hello, <b>APL</b>ers!</p>'
 hr.Caption←'Form with Button'
 hr.(Coord Size Posn)←'Pixel'(300 300)(20 20)
 hr.HTML,←'<form action="#"><button>Click Me!</button></form>'
 hr.onHTTPRequest←'my_callback'
 hr.InterceptedURLs←1 2⍴'*' 1  ⍝ intercept all requests
```

The form should look like this:



If you click on the button, the content should be replaced:



2.4 Using `HttpUtils` with HTMLRenderer

`HttpUtils` is a utility namespace provided with Dyalog APL v16.0 and later. It contains classes and functions for processing and formatting HTTP request and response messages. `HttpUtils` is designed to work with HTMLRenderer and with Conga[1] 'HTTP' mode.

`HttpUtils` is distributed in the /Library/Conga/ folder in your Dyalog installation and can be loaded using the SALT Load command. Both of the following statements will load HttpUtils, though the latter is suitable for running under program control.

```
      ]load HttpUtils
⍝SE.SALT.Load 'HttpUtils'
```

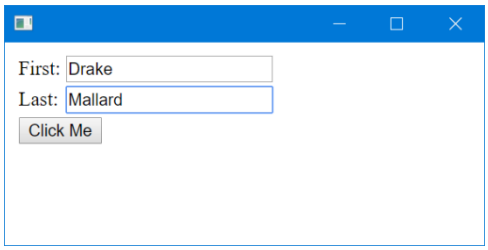
`HttpUtils` is maintained in the library-conga Dyalog GitHub repository found at https://github.com/Dyalog/library-conga. There you can see the revision history and you may participate in the development community by reporting issues and by posting questions and suggestions.

The following example shows a simple HTML form with 2 input fields and a submit button. The callback is processed using the `HttpRequest` and `HttpResponse` classes found in `HttpUtils`.

[1] Conga is the Dyalog TCP/IP utility library – HTTP mode was introduced in Conga version 3.0

```
     ∇ r←SimpleForm args;evt;html;req;resp;who;iurls
[1]    :If 0∊⍴args ⍝ Setup
[2]        html←'<form method="post" action="SimpleForm"><table>'
[3]        html,←'<tr><td>First: </td><td><input name="first"/></td></tr>'
[4]        html,←'<tr><td>Last: </td><td><input name="last"/></td></tr>'
[5]        html,←'<tr><td colspan="2"><button>Click Me</button></td></tr>'
[6]        html,←'</table></form>'
[7]        evt←'Event' 'HTTPRequest' 'SimpleForm'
[8]        iurls←'InterceptedURLs'(1 2⍴'*' 1)
[9]        'hr'⎕WC'HTMLRenderer'('HTML'html)('Coord' 'ScaledPixel')('Size'(200 400))iurls evt
[10]   :Else ⍝ handle the callback
[11]       req←⎕NEW #.HttpUtils.HttpRequest args ⍝ create a request from the callback args
[12]       resp←⎕NEW #.HttpUtils.HttpResponse args ⍝ create a response from the callback args
[13]       who←req.(FormData⍪.Get)¨'first' 'last' ⍝ req.FormData has the data from the form
[14]       who←⊂' ',¨who
[15]       resp.Content←'<h2>Welcome ',who,'!</h2>' ⍝ set the content for the response
[16]       r←resp.ToHtmlRenderer ⍝ convert the response to HTMLRenderer format
[17]   :EndIf
     ∇
```

Running `SimpleForm ''` displays the form. After filling in the form and clicking the button, SimpleForm is called again as the callback function for the HTTPRequest event, but this time `args` is non-empty and the callback portion lines [11-16] are executed.




```
[11]       req←⎕NEW #.HttpUtils.HttpRequest args   ⍝ create a request from the
callback args
```

The `HttpRequest` constructor accepts an argument of HTMLRenderer callback data and will parse and extract the various bits of the HTTP message into a more useful and accessible format.

```
[12]       resp←⎕NEW #.HttpUtils.HttpResponse args ⍝ create a response from the
callback args
```

We create a response object to send back to HTMLRenderer. Like `HttpRequest`, the `HttpResponse` constructor also accepts an argument of the HTMLRenderer callback data.

```
[13]       who←req.(FormData⍪.Get)¨'first' 'last'   ⍝ req.FormData has the data
from the form
```

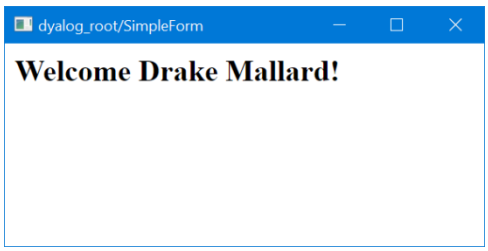
The `HttpRequest` class has extracted the HTML form field values into `FormData`. The values are retrievable by their field names in the HTML form, in this case 'first' and 'last'. Refer to lines `[3-4]` in `SimpleForm` to see where the field names were originally assigned.

```
[14]       who←⊂' ',¨who
[15]       resp.Content←'<h2>Welcome',who,'!</h2>' ⍝ set the content for the
response
```

We now set `Content` in the response to be our new content for the page. The default content type is 'text/html', but other content types can be specified as appropriate for your application.

```
[16]       r←resp.ToHtmlRenderer                   ⍝ and send it back
```

Finally, the response's `ToHtmlRenderer` method formats and populates a result appropriate for the callback and our friendly message is displayed.


dyalog_root/SimpleForm
Welcome Drake Mallard!

3 Generating HTML

To use the HTMLRenderer, you either need to be able to produce HTML and associated documents, or allow HTTP requests to pass through, as demonstrated in the example in chapter 1.

Dyalog provides a number of tools to help you generate HTML.

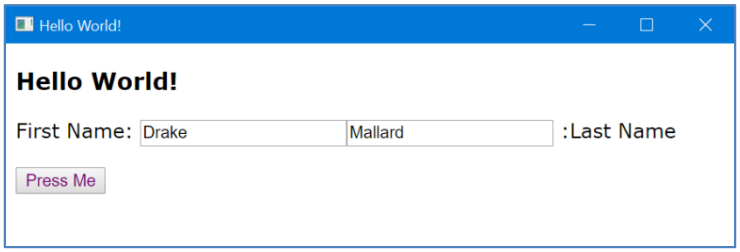
SharpPlot

The SVG data produced by the RenderSVG method can be assigned directly to the HTML property of an HTMLRenderer object. The CEF accepts SVG in place of HTML and is able to render it without further intervention. You can also use the various Save… functions in SharpPlot to save graphs in SVG or other formats, and link to them using an HTML `img` tag.

DUI – Dyalog User Interface Utility Library

DUI is an evolving library to assist in creating HTML content. Originally a part of MiServer, DUI is designed to enable you to create HTML that can be run locally with HTMLRenderer or on the net with MiServer – without changing your code. DUI contains APL code that is able to generate HTML, CSS and JavaScript based widgets based on the HTML5 widget set, Syncfusion controls (which are bundled with Dyalog), jQueryUI and other third-party widgets. DUI is currently available from the Dyalog GitHub repository at https://github.com/Dyalog/DUI. To use DUI, you will need to download or clone the repository. To illustrate the DUI style of coding, the following code should produce a form with two input fields and a button:

```
]load /path_to_DUI/DUI
DUI.Initialize
page←⎕NEW Page
page.Add _.title 'Hello World!'
page.Add _.Style 'body' ('font-family' 'Verdana')
page.Add _.h3 'Hello World!'
form←page.Add _.Form
'fn' form.Add _.Input 'text' 'Drake' 'First Name: '
'ln' form.Add _.Input 'text' 'Mallard' ' :Last Name' 'right'
p1←'p1' form.Add _.p ''
b1←'b1' form.Add _.Button 'Press Me'
b1.style←'color:purple'
page.Size←200 600
```



4 Technical Overview

The HTML Renderer is implemented using the Chromium Embedded Framework (CEF); for more information on CEF visit https://en.wikipedia.org/wiki/Chromium_Embedded_Framework.

4.1 HTMLRenderer on non-Windows platforms

The HTMLRenderer on non-Microsoft Windows platforms is an X-Windows application. As such there are a set of pre-requisites that are needed on the operating system instance on which the Dyalog interpreter is running (this in X-Windows terms is the server) and a set of pre-requisites that are needed on the operating system instance where the output will be displayed (in X-Windows terms the client). In most cases these two sets of functionality run in the same operating system instance. However, this means that a typical non-GUI installation of a Linux distribution is unlikely to allow you to create an HTMLRenderer object even if you are trying to display it elsewhere by setting the DISPLAY variable appropriately.

For Linux, we have tried creating the HTMLRenderer on a number of common distributions and versions. See https://forums.dyalog.com/viewtopic.php?f=20&t=1505 which details what pre-requisites are needed for the HTMLRenderer on those distributions.

If you get a LIMIT ERROR when attempting to create an HTMLRenderer object and you are either using a distribution/version that is not in the list below, or have ensured that you have met the pre-requisites mentioned below, then run the following expression from within Dyalog APL:

```
)sh ldd $DYALOG/lib/htmlrenderer.so | grep found
```

This should list any missing pre-reqs. Please let Dyalog know so that we can update the supported versions matrix.

Note: As of 2019-05-10, Dyalog does not support the HTMLRenderer being used on a Virtual Machine in which Linux has been installed. We are attempting to understand why this does not run reliably, but the issue may be intractable.

5 Dyalog GUI Implementation for HTMLRenderer

This section highlights specific aspects the HTMLRenderer. For a complete description of the Properties, Events and Methods for the HTMLRenderer object, please refer to the object reference at http://help.dyalog.com/17.1/Content/GUI/Objects/HTMLRenderer.htm

5.1 Properties

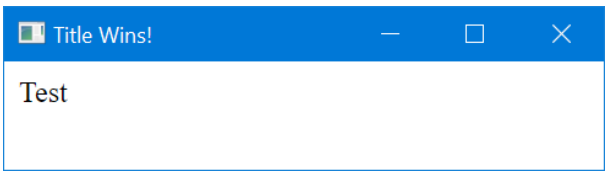
As HTMLRenderer is an object in the Dyalog GUI framework, it has many of the expected properties for a ⎕WC GUI control. The properties for HTMLRenderer are found in table 1, with properties specific to HTMLRenderer highlighted in red.

***Table 1**. HTMLRenderer properties*

Type	HTML	Posn
Size	URL	Coord
Border	Visible	Event
Sizeable	Moveable	SysMenu
MaxButton	MinButton	IconObj
Data	Attach	Translate
KeepOnClose	AsChild	InterceptedURLs
CEFVersion	Caption	MethodList
ChildList	EventList	PropList

Note that the caption appearing in the title bar of the HTMLRenderer window can be set either with the `Caption` property or by a <title> element within the <head> element in the HTML for the page. If both are set, the <title> element takes priority. For example:

```
html←'<head><title>Title Wins!</title></head><body>Test</body>'
'hr' ⎕WC 'HTMLRenderer' ('HTML' html)('Caption' 'Cap Wins!')('Size' (10 20))
```



5.2 Properties Specific to HTMLRenderer

HTML

The `HTML` property is a character vector of the content rendered in the object. The interpreter does not perform any pre-processing of the text. As such, it must be properly formed HTML using single-byte (⎕DR 80 or 82) character data, including any necessary escaping and encoding.

NOTE: Typically, you will need to UTF-8 encode any text outside the Unicode range 0-127.

URL

The URL property is a character vector representing the "root" URL of the object. If not specified, 'dyalog_root' is the default value of URL. If subsequent requests for resources are received via the HTTPRequest event, the URL element of the event's arguments can be examined to see if it begins with the "root". If so, the content is intended to be provided locally by your application, otherwise, it should be retrieved from the URL element of the argument.

AsChild

This property only has an effect on Microsoft Windows platforms.

The AsChild property is a Boolean indicating how the HTMLRenderer object should be treated. Possible values are:

- 1 – the HTMLRenderer object should be treated as a child of its parent object.
- 0 – the HTMLRenderer object should be treated as a top level object similar to how a Form object is treated.

The default is 0.

InterceptedURLs

The InterceptedURLs property is a 2-column matrix that specifies whether HTMLRenderer will trigger an HTTPRequest event for the requested URL. The first column is a wild-carded character scalar or vector containing a pattern to match. The second column is a Boolean indicating whether HTMLRenderer should trigger an HTTPRequest event for a URL matching the corresponding pattern. InterceptedURLs may contain any number of rows. The default is 0 2⍴'' meaning that no URLs will trigger an HTTPRequest event.

Examples:

The following will trigger an HTTPRequest event for all requested URLs

```
InterceptedURLs ← 1 2⍴'*' 1
```

The following will attempt to retrieve from the net URLs containing '.dyalog.com' and trigger an HTTPRequest event for all other requested URLS

```
InterceptedURLs ← 2 2⍴'*.dyalog.com*' 0 '*' 1
```

CEFVersion

Returns version information about the CEF. This is used primarily for support and debugging purposes.

5.3 Events

The events for HTMLRenderer are found in table 2, with events specific to HTMLRenderer highlighted in red.

***Table 2**. HTMLRenderer events*

Close	Create	HTTPRequest
WebSocketUpgrade	WebSocketReceive	WebSocketClose
WebSocketError	DoPopup	SelectCertificate

5.4 Events Specific to HTMLRenderer

HTTPRequest

An `HTTPRequest` event is raised whenever content is required that is not provided by the HTML property. This could be generated by a form submission, clicking on a hyperlink, an AJAX request or a link to a resource like a stylesheet, image or JavaScript file.

The event message reported as the result of □DQ or supplied as the right argument to your callback function, is a 11-element vector as described in table 3.

NOTE: the event message only had 10 elements in version 16.0. Application code should not assume a specific length for this, or indeed any other event messages.

***Table 3**. Explanation of the 11-element vector HTTPRequest event message*

`[1]`	`HTMLRenderer object name or reference`
`[2]`	`Event name 'HTTPRequest'`
`[3]`	`Constant 'ProcessRequest'`
`[4]`	`0`
`[5]`	`0`
`[6]`	`''`
`[7]`	`''`
`[8]`	`Requested URL`
`[9]`	`HTTP Request Headers`
`[10]`	`HTTP Request Body`
`[11]`	`HTTP Method (new in version 17.0): typically 'GET' or 'POST', but other methods may occur.`

When preparing a response, elements of the event message need to be updated. Specifically:

- `[4]` : set to `1` if you will handle the request by updating other elements of the event message.

  In a typical scenario, you will check whether the requested URL in [8] begins with the "root" URL:
  - If it does, then your application will supply the content of the response. In this situation, update the appropriate elements of the event message, setting element `[4]` to 1 and return.
  - If it does not, then the request is for some external resource. Return without changing any elements of the event message and HTMLRenderer will attempt to retrieve the requested resource.

- `[5]` : set to the HTTP status code for the response. Success is indicated by code `200`.

- `[6]` : set to the HTTP status message for the response. Success is indicated by the message `'OK'`.

- [7] : set to the MIME type of the response. For sending HTML, the MIME type is 'text/html'.
- [9] : set to any HTTP message headers necessary for the response.
- [10] : set to the body of the response. Typically this will be HTML.

WebSocketUpgrade, WebSocketReceive, WebSocketClose, WebSocketEnd

Please refer to Section 6, *Websocket support* for more information.

DoPopup

A DoPopup event is raised whenever the CEF client executes a request for a new window to be opened. This would typically be when a link element specifies a target attribute of "_blank", as in:

```
<a href="http://www.dyalog.com" target="_blank">
```

When a DoPopup event occurs, the application should inspect the request and open another HTMLRenderer as appropriate.

SelectCertificate

A SelectCertificate event is raised whenever a resource is requested from a server that requires a certificate for security. The available certificates are in element [7] of the callback arguments. The application should select one of the certificates and set element [3] to its origin-0 index in the Certificates element.

Elements of the SelectCertificate callback arguments

[1]	HTMLRenderer object name or reference
[2]	Event 'SelectCertificate' or 848
[3]	Certificate index (result only)
[4]	Host address
[5]	Host port
[6]	'is proxy'
[7]	Certificates (see below)

Certificates is a vector of namespaces, each of which represents a certificate and contains the following variables:

Name	Description
DER	The DER-encoded certificate
Subject	A namespace containing variables CommonName, CountryName and DisplayName for the certificate subject.
Issuer	A namespace containing variables CommonName, CountryName and DisplayName for the certificate issuer.
SerialNumber	Character vector certificate serial number

5.5 Methods

The methods for HTMLRenderer are found in table 4, with events specific to HTMLRenderer highlighted in red.

***Table 4**. HTMLRenderer methods*

Detach	PrintToPDF	WebSocketSend
WebSocketClose	ShowDevTools	Wait

5.6 Methods Specific to HTMLRenderer

WebSocketSend, WebSocketClose
Please refer to section 6, WebSocket Support, later in this document.

ShowDevTools
The ShowDevTools method is used to open or close the Chromium Developer Tools console as in:

```
'hr' □WC 'HTMLRenderer'
hr.ShowDevTools 1  ⍝ display developer tools
hr.ShowDevTools 0  ⍝ hide developer tools
```

6 Websocket support

6.1 Websocket Overview

In a typical HTTP application, all communication originates from the client which sends requests to the server which in turn sends back a response. When an application wanted to "push" information from the server to the client, the typical way to fake this was to have the client periodically poll the server so the server could send back any information that it had to offer. With the use of websockets, true asynchronous, bi-directional transmission between the client and server is possible.

HTMLRenderer presents a straightforward API to use websockets. A typical scenario would look something like this:

1. The client initiates an HTTP "upgrade" request to the server. After some validation and handshaking with the server, the websocket is established. With HTMLRenderer, the validation and handshaking are currently done behind the scenes and by the time you receive a WebSocketUpgrade event, the websocket already established.

2. Once the websocket is established, either the client or the server can send information which will trigger a "receive" event on the other end. No response is expected as a part of the websocket protocol. Whatever response you send (or don't) is up to your specific application.

3. Either side can close the websocket.

4. Websocket error events may be triggered when an unexpected error, like disruption in the connection, occurs.

JavaScript in the CEF client		**HTMLRenderer in the workspace**
ws = new websocket(url); Initiate the request	→	WebSocketUpgrade event The websocket is established
ws.send("message");	→	WebSocketReceive event
ws.onmessage event	←	WebSocketSend method
ws.close()	→	WebSocketClose event
ws.onclose event	←	WebSocketClose method
ws.onerror event is triggered when there is some error like the connection going down		WebSocketError event occurs when there is some error like the connection going down

The client may request multiple upgrades resulting in multiple websockets, each with its own unique id.

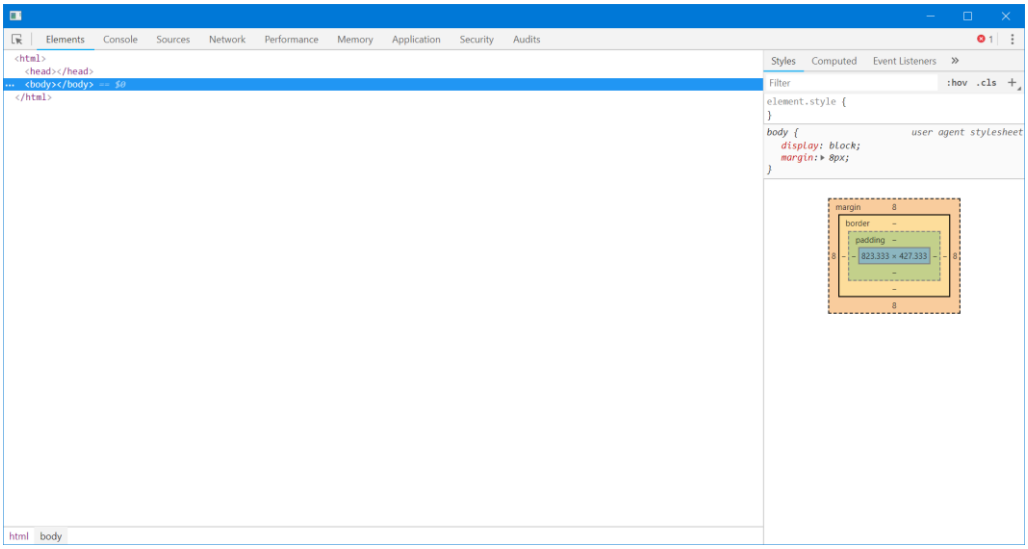
7 Debugging HTMLRenderer

Chromium's developer tools can be used to inspect and debug the rendered HTML content by calling the `ShowDevTools` method with an argument of 1.

For example:

```
'hr' □WC 'HTMLRenderer'
hr.ShowDevTools 1
```

Will bring up the developer tools window similar to:



8 Running HTMLRenderer under a Windows Runtime Application

To run HTMLRenderer under a Windows runtime interpreter (dyalogrt.exe) you should:

1. Create your runtime environment as described in the Dyalog for Microsoft Windows Installation and Configuration Guide
2. Copy the following files from the Dyalog installation folder into the same folder as the dyalogrt.exe:

   htmlrenderer.dll
   cef.pak
   cef_100_percent.pak
   cef_200_percent.pak
   cef_extensions.pak
   cef_sandbox.lib
   chrome_elf.dll
   d3dcompiler_43.dll
   d3dcompiler_47.dll
   devtools_resources.pak
   icudtl.dat
   libEGL.dll
   libGLESv2.dll
   libcef.dll
   libcef.lib
   locales
   natives_blob.bin
   snapshot_blob.bin
   swiftshader
   v8_context_snapshot.bin

9 Resources and References

The Dyalog webinar "*Something Old, Something New & Something Experimental*" includes a discussion and demonstration of the HTMLRenderer; it can be viewed at https://dyalog.tv/webinar.

Code samples can be copied-and-pasted from an HTML version of this document at http://docs.dyalog.com/17.1/HTMLRenderer User Guide.htm.