So far we have identified the specific Seaside messages to create particular HTML constructs in an ad-hoc manner as needed for particular features. Now we will attempt a more systematic approach by looking at the various pieces of an HTML document and how Seaside generates the pieces.

An HTML document consists of a series of nested elements. An element can be a single, self-closing, tag (such as '
' for a line break) or a pair of tags (such as '<h1>Seaside</h1>' for a level-one heading) enclosing text and/or embedded elements. Depending on the tag type, the open tag may contain attributes that further describe the element (such as 'GLASS' where 'href' is the attribute name and the URL is the attribute value). The outermost element in an HTML document is the 'html' element, and it contains one 'head' and one 'body' element.

The primary role of any web framework is to generate HTML pages to be rendered on the user's browser. In Seaside, the HTML page is represented by one or more subclasses of WAComponent and methods may be implemented in the subclass to provide content to both the head and the body.

The component adds content to the page's head element by implementing a method 'updateRoot:' which is passed an instance of WAHtmlRoot. The head element must contain a 'title' element, so Seaside provides a default of 'Seaside' that can be overridden by sending 'title:' to the object passed to the 'updateRoot:' method. Other elements in the head are optional and include meta data and references to external CSS and JavaScript files (using the 'link' element). All components have an opportunity to add information to the page's head element before any of the body is rendered. The tree of components is identified by sending the message 'children' to the root component (so if you fail to implement the 'tree' method and include subcomponents, then the subcomponents will not have an opportunity to contribute to the head element).

The body element is the main place things are displayed on a HTML document. A component adds content to the page's body element by implementing a method 'renderContentOn:' which (by default) is passed an instance of WARenderCanvas. The component can add content to the page by sending messages to the html canvas received as an argument to this method.

To add text to a page, send a string as an argument with the 'text:' message to the html canvas. The string will be encoded properly so that special characters (such as '<' that would otherwise be interpreted as the beginning of a tag) are displayed on the browser. To add raw HTML code to a page, use the 'html:' message to avoid any encoding.

To add an element to a page, send a unary message to the html canvas identifying the element type desired, and then send messages to the resulting "brush" (representing the element) to (1) add attributes to the element and (2) embed content in the element. To embed content, send the 'with:' message providing as an argument either a block (that will be evaluated) or another object (that will be converted to a string, by default with 'displayString' via 'greaseString'). For example, to add an emphasis element with the text 'very' ('very'), you could use the following code.

```
canvas emphasis with: 'very'.
```


As a shortcut for cases (like the above) in which no attributes need to be set, there are a parallel set of keyword messages that take one argument that is passed with the 'with:' message.

```
canvas emphasis: 'very'.
```

Code blocks are generally used as the argument to a 'with:' message to embed other elements inside an element, though you could, of course, use a code block where text would be sufficient. The following is equivalent to the above two examples.

```
canvas emphasis: ['very'].
```

The HTML 4.01 Specification (http://www.w3.org/TR/REC-html40/) contains an index of 91 elements (http://www.w3.org/TR/REC-html40/index/elements.html), ten of which are deprecated. In Seaside, WARenderCanvas (and its superclass, WAHtmlCanvas) provides support for most of the non-deprecated elements through methods identified in the following table. As you can see, the Seaside approach is that a full message name is preferred over an abbreviation. This follows the general Smalltalk philosophy that code is more often read than written and spelling things out imposes less mental effort on the reader.

HTML Element Name	Seaside Message	Smalltalk Class (if not WAGenericTag)
A	anchor	WAAnchorTag
ABBR	abbreviated	
ACRONYM	acronym	
ADDRESS	address	
BIG	big	
BLOCKQUOTE	blockquote	
BR	break	WABreakTag
BUTTON	button	WAButtonTag
CAPTION	tableCaption	
CITE	citation	
CODE	code	
COL	tableColumn	WATableColumnTag
COLGROUP	tableColumnGroup	WATableColumnGroupTag
DD	definitionData	
DEL	deleted	WAEditTag
DFN	definition	
DIV	div	WADivTag
DL	definitionList	
DT	definitionTerm	
EM	emphasis	
FIELDSET	fieldSet	WAFieldSetTag
FORM	form	WAFormTag
H1	heading	WAHeadingTag
H2	heading	WAHeadingTag
H3	heading	WAHeadingTag
H4	heading	WAHeadingTag
H5	heading	WAHeadingTag

Name	Seaside	Description
H6	heading	WAHeadingTag
HR	horizontalRule	WAHorizontalRuleTag
IFRAME	iframe	WAIframeTag
IMG	image	WAImageTag
INPUT	cancelButton	WACancelButtonTag
	checkbox	WACheckboxTag
	fileUploader	WAFileUploadTag
	imageButton	WAImageButtonTag
	multiSelect	WAMultiSelectTag
	passwordInput	WAPasswordInputTag
	radioButton	WARadioButtonTag
	submitButton	WASubmitButtonTag
	textInput	WATextInputTag
INS	inserted	WAEditTag
KBD	keyboard	
LABEL	label	WALabelTag
LEGEND	legend	
LI	listItem	
MAP	map	WAImageMapTag
OBJECT	object	WAObjectTag
OL	orderedList	WAOrderedListTag
OPTION	option	WAOptionTag
OPTGROUP	optionGroup	WAOptionGroupTag
P	paragraph	
PARAM	parameter	WAParameterTag
PRE	preformatted	
Q	quote	
SAMP	sample	
SCRIPT	script	WAScriptTag
SELECT	select	WASelectTag
SMALL	small	
SPAN	span	
STRONG	strong	
SUB	subscript	
SUP	superscript	
TABLE	table	WATableTag
TBODY	tableBody	
TD	tableData	WATableDataTag
TEXTAREA	textArea	WATextAreaTag
TFOOT	tableFoot	
TH	tableHeading	WATableHeadingTag
THEAD	tableHead	
TR	tableRow	
TT	teletype	
UL	unorderedList	WAUnorderedListTag
VAR	variable	

There are several elements that Seaside does not directly support (but could be included using a generic tag by sending #'tag:').

- The following elements are deprecated in the HTML specification: applet, basefont, center, dir, font, isindex, menu, s strike, and u.
- The b and i elements are related to styling that should be done with CSS.
- The following elements are structural or appear only in the head section so are provided using other mechanisms: base, body, head, html, link, meta, style, and title.
- The remaining elements that have no direct Seaside support are the following: area, bdo, frame, frameset, noframes, and noscript.

Once you have identified a tag for an element, there might be attributes to set on that tag. The following core attributes are available on most elements by sending a message to the element object with a string argument:

- id (#'id:') – a document-wide unique identifier for the element. Because Seaside components can be embedded in other components, it is generally not a good idea to hard-code an identifier. When possible, send the message #'nextId' to the instance of WARenderCanvas passed to #'renderContentOn:' in order to get a unique identifier. Alternatively, you can send #'ensureId' to a brush.
- class (#'class:') – a space-separated list of class names. This attribute is primarily used to associate CSS style with an element.
- style (#'style:') – a style declaration for the element. It is considered a better practice to use an external style sheet.
- title (#'title:') – advisory information about the element. Visual browsers will often show this as a tool tip.

The following attributes are available on most elements:

- lang (#'language:') – the base language of an element's attribute values and text content.
- dir (#'direction:') – directionality of text. Acceptable values are 'ltr' for left-to-right text and 'rtl' for right-to-left text.

The following event-related attributes can be set on most elements and will trigger a script (typically JavaScript) in the browser:

- onclick (#'onClick:') – a pointer button was clicked.
- ondblclick (#'onDoubleClick:') – a pointer button was double clicked.
- onmousedown (#'onMouseDown:') – a pointer button was pressed down
- onmouseup (#'onMouseUp:') – a pointer button was released
- onmouseover (#'onMouseOver:') – a pointer was moved onto
- onmousemove (#'onMouseMove:') – a pointer was moved within
- onmouseout (#'onMouseOut:') – a pointer was moved away

- onkeypress (#'onKeyPress:') – a key was pressed and released
- onkeydown (#'onKeyDown:') – a key was pressed down
- onkeyup (#'onKeyUp:') – a key was released

Other attributes tend to be element-specific and can be found in the element's class (shown in the third column in the above table). Now we will look at some of the various classes and methods involved in the many steps from starting a web server to accepting a HTML request on a socket to sending an HTML response.

- The Seaside Control Panel (one if the three initial windows in the Seaside One-Click Experience) provides a graphical user interface over the default instance (singleton) of WAServerManager that manages a collection of instances of subclasses of WAServerAdapter.
- Initially, there is one adapter, an instance of WAComancheAdaptor, which is configured to listen on port 8080 and (by default) pass requests to the default instance (singleton) WADispatcher.
- WAComancheAdaptor>>#'basicStart' creates an HttpService on the provided port.
- HttpService>>#'runWhile:' creates a TcpListener that calls back to HttpService>>#'value:' with each newly accepted socket.
- HttpService>>#'serve:' creates an instance of HttpAdaptor and passes it the socket and itself.
- HttpAdaptor>>#'beginConversation' reads from the TCP socket and instantiates an instance of HttpRequest. This request is passed to HttpAdaptor>>#'dispatchRequest:' to get an instance of HttpResponse that is returned on the socket.
- HttpAdaptor>>#'dispatchRequest:' calls HttpService>>#'processHttpRequest:' on the instance first created above and it calls WAComancheAdaptor>>#'processHttpRequest:' on the instance first created above.
- WAComancheAdaptor>>#'processHttpRequest:' calls #'process:' on itself with the HttpRequest.
- WAServerAdaptor>>#'process:' creates an instance of WARequestContext (containing an instance of WARequest and WABufferedResponse).
- WAServerAdaptor>>#'handleRequest:' passes the instance of WARequest and to WADispatcher>>#'handleRequest:'. The result is an instance of WAResponse that is converted to an instance of HttpResponse that is eventually passed back to the client browser as a web page.

As described above, WAComancheAdaptor creates an instance of WARequest and passes it to the default WADispatcher that is responsible for finding someone to create a WAResponse to return to WAComancheAdaptor.

- There is a top-level dispatcher returned by the class-side method, #'default', which selects an instance of a subclass of WARequestHandler to receive the request. Of course, in order for an application to be found, an instance of WAApplication must have been registered. This is what is done when you send #'register:asApplicationAt:' to WAAdmin with your subclass of WAComponent and a string. The string you pass (such as 'hello') is used to define a path on your web server (such as 'http://localhost:8080/hello').

- The dispatcher sends the instance of WARequest to WAApplication>>#'handleRequest:' that looks for an instance of WASession to handle the request. The lookup is done using the '_s' field included in the URL or in a cookie (depending on a configuration setting). If the session key is not found (perhaps it expired and has been removed from the cache), is not provided, or has expired then a new session is created.
- WARegistry>>#'dispatch:to:' calls WASession>>#'handle:', which is implemented in WARequestHandler and eventually calls #'handleFiltered:' on itself (an instance of WASession or a subclass).
- The #'handleFiltered:' method looks at the passed-in URL fields for an 'action key' (the one with the key '_k'). If such a field is found in the request then the value is used as a lookup into the session's continuation dictionary. If there is a continuation available with the key, then it is evaluated with the request. We will look more at that below. First we look at the handling of an initial request.
- If the request does not include an 'action key', then the #'start:' message is sent to the session. If the request includes an 'action key' but there is no continuation for that key, then #'unknownRequest' is called which, by default, calls #'start:'.
- WASession>>#'start:' looks for the application's configuration preference for a 'mainClass' (by default, this is the class WARenderLoopMain) and calls #'start' on a new instance.
- WARenderLoopMain>>#'start' creates a new instance of the application's renderPhaseContinuationClass (by default, WARenderPhaseContinuation), and sends #'initialRequest:' to each visible presenter (the root component and its children). This is an opportunity to capture any parameters (e.g., to simulate a RESTful application) and redirect or configure the initial page based on these parameters.
- Once the initialRequest: messages have been sent, WARenderLoopMain>>#'start' calls #'captureAndInvoke' on the continuation. After some setup, WARenderPhaseContinuation>>#'processRendering:' creates the document (typically an instance of WAHtmlDocument).
- At this #'updateUrl:' is sent to each presenter. This gives them an opportunity to modify the base URL used when rendering.
- Next, #'updateRoot:' is sent to each presenter. This allows you to customize the <head> element of the HTML document. Typically, you would update the title and add CSS and JavaScript links here.
- Finally, #'renderWithContext:' is called on the root-level component (the one that was registered as the application). This uses WARenderVisitor to call your component's #'renderContentOn:' method.

If the request does not result in #'start:' being sent to the session, then we are dealing with a continuation created by an earlier page. Let's walk through how that continuation is created and then used.

- When rendering an HTML document, one might add an anchor element to the page and give the anchor a callback by sending #'callback:' to an instance of WAAnchorTag. For example:

```
renderLogoutAnchorOn: canvas

   canvas anchor
      callback: [self logout];
      with: 'Logout'.
```

- Sending the #'callback:' message will invoke WAAnchorTag>>#'callback:' which can be refactored as follows to allow easier discussion.

```
WAAnchorTag>>#callback: aNiladicValuable

      | callback id |
"4"   aNiladicValuable argumentCount > 0 ifTrue: [
"5"      GRInvalidArgumentCount signal: 'Anchors expect a niladic callback.'
"6"   ].
"7"   callback := WAActionCallback on: aNiladicValuable.
"8"   id := self storeCallback: callback.
"9"   self url addField: id.
```

- Line 4 checks to ensure that the block does not expect any arguments and line 5 provides an error in this case.
- Line 7 wraps the block in an instance of WAActionCallback.
- Line 8 saves the callback in an instance of WACallbackRegistry associated with the current session and returns a unique key.
- Line 9 adds the unique key to the URL that will be associated with the anchor tag.
- When the page is rendered, the instance of WAAnchorTag will add itself to the page as an <a> element with an 'href' attribute of the new URL. In the following URL, there is a parameter '4' that can be used to find the WAActionCallback holding the block.

```
http://localhost:8080/boquitas?_s=cDBBHwMjSSQFwGAD&_k=8k2j2bSW&4
```

- When the user clicks on the anchor an HTTP GET request is submitted to the server. The server dispatches it to the application registered with the name 'boquitas'. The application then looks for a session with the key 'cDBBHwMjSSQFwGAD' and passes the request to the session. The session then looks for a 'continuation' with the key '8k2j2bSW' and passes the request to the instance of WAActionPhaseContinuation built during the earlier page rendering.
- WAActionPhaseContinuation>>#'runCallbacks' then invokes WACallbackRegistry>>#'handle:' to find and evaluate the appropriate callbacks based on the unique key added to the URL.

To learn more about this process, select one of the methods listed above and add 'self halt.' to the code.

