The IMHO Application Server

A Tutorial and Reference Manual

Jesse Bouwman

Craig Brozefsky

The IMHO Application Server: A Tutorial and Reference Manual
by Jesse Bouwman

by Craig Brozefsky

Table of Contents

List of Tables

List of Examples

Chapter 1. Overview

IMHO is a system for writing applications that are served over the Internet, and manipulated using a web browser. It provides mechanisms to solve some of the common issues that arise in writing these types of application. For instance, IMHO provides simple ways of maintaining state between requests, of inserting and extracting values from form elements, and assembling interface pages from components. Beyond concrete things such as these, IMHO's design has been motivated by the desire for a web toolkit that is simple, self-consistent, predicatable, and a pleasure to use.

Most of IMHO is written in ANSI Common Lisp, and it takes full advantage of the macro facility and object system.

IMHO runs inside of a Lisp process, and communicates with an Apache web server using the Apache Group's JServ module.

Chapter 2. Applications and Sessions

IMHO provides state to HTTP served applications by creating session objects for application clients[1]. When a client makes a first request of IMHO, the server generates a random session identifier which becomes part of the URL for all session-sensitive components of the application. AN IMHO URL tends to look like the following:

Example 2-1. An IMHO URL

```
http://rapunzel.onshore.com/apps/application-1/chkjdiuwefhkjchvs/C1000/display?001
http://[host]/[adapter]/[application]/[session]/[component]/[method]?[arg1]@...
[argn]
```

All intra-application links .. part of their URLS is a session key, a random identifier that it used for looking up the session into which the link refers, what URL is requested.

Notes

1. At present, IMHO ensures that there is a session for all clients. Session creation really ought to be optional, and take place at the point a ’session’ is really needed.

Chapter 3. Objects

3.1. Defining Elements

All objects that can be displayed on a client browser in IMHO derive in some way from `html-element`. IMHO objects are ordinary *CLOS* objects, and can be defined, redefined, and created using standard CLOS syntax. Classes for basic elements can be defined by subclassing `html-element`. Once a class has been defined, it can be used to 'display' part of a response to an HTTP request. Here's a very simple IMHO application:

```
(defclass hello-world (html-element)
  )

(defmethod render-html ((element hello-world) stream)
  (format stream "Hello World!"))

(defapplication hello-world-app
  :base-url "hello"
  :initial-element hello-world)
```

If the JServ adaptor is configured to serve applications under "/imho" (the so-called 'mount point'), then once running, this application could be invoked at a URL that looks like this:

```
http://127.0.0.1/imho/hello
```

There are only a few things necessary to set an application running: you need to specify where it can be found ('base-url'), and what element ought to be displayed when the application receives its first request from a client. These two bits of information are declared using 'defapplication'. Once that's done, the only remaining thing to do is to tell imho to start serving the application, by invoking: **`(imho:init/application 'hello-world-app :start)`**

> **Note:** The first time you start up any IMHO application, IMHO will attempt to bind to `imho::*lisp-server-port*` in order to recieve requests from your JServ adaptor. If you started a Java VM to handle JServ requests, you should kill it, or move it to another port, or configure your JServ mount point for IMHO and `imho::*lisp-server-port*` to use a different port for their communications.

This main element of this application, 'hello-world', has the single requisite superclass, html-element, but no other interesting characteristics. The only other code required is a 'render-html' method, to provide the element with a means of displaying itself. 'render-html' is the function that is called whenever IMHO is asked to display a component. To provide 'render-html' for 'hello-world', you define a method for this function, where the first parameter is of type 'hello-world', and the second is a variable that will be bound to the response stream at the time it is displayed.

To provide display behavior via a template, a file named "hello-world.html", containing the single line:

```
Hello World!
```

must exist in the template directory of the application.

3.2. Creating Instances

IMHO html-elements may be created using the conventional CLOS make-instance methods. An additional constructor is provided, session-instance.

Chapter 4. The IMHO Package

4.1. Classes

This section describes the basic classes that must be manipulated in order to build working IMHO programs.

4.1.1. The `application` Class

The `application` encapsulates information common to all sessions of an application, such as the path to templates for rendering its html elements, and the application's public base URL. In general, you don't need to directly subclass application (using defclass), you can use the 'defapplication' macro.

```
defmacro defapplication (name &key initial-element initial-method  session-class base-url
```

4.1.2. The `http-session` Class

This class provides the relation of client requests to server-side state.

```
(defclass http-session ()
  ((session-id
    :accessor session-id
    :initarg :id)
   (session-html-elements
    :accessor session-html-elements
    :initform (make-hash-table :test 'equal))
   (session-instances
    :accessor session-instances
    :initform (make-hash-table :test 'equal))
   (session-application
    :accessor session-application
    :initarg :application
    :initform nil)
   (active-response
    :initform nil)
   (last-url
    :accessor last-url
    :initarg :last-url
    :initform nil
    :documentation
    "The last URL visited by this session's client.  This is really
here to support a 'go back' link from a help system page. I wonder if
this is the right way to do it.")
   (help-target
    :accessor help-target
    :initarg :help-target
    :initform "help-main")
   (timeout
    :accessor session-timeout
    :initarg :session-timeout
```

```
    :initform 300
    :documentation
    "Idle Timeout in seconds")
   (timestamp
    :accessor session-timestamp
    :initarg :session-timestamp
    :initform (get-universal-time)
    :documentation
    "Used for determining if session has timed-out"))
  (:documentation
   "A session encapsulates all required information about a set of
interactions with a client browser.  Subclasses should store
authentication data and other objects that persist across requests."))
```

4.1.3. HTML elements

```
;; ---------------------------------------------------------------
;; framework class: html-element
;;
;; This is the root of the html-element inheritance graph.
;;
;; Some classes to derive their rendering behavior from HTML templates
;; residing in the filesystem, and others from overriding
;; 'render-html'.
(defclass html-element ()
  ((element-external-name
    :reader element-external-name
    :initarg :element-external-name
    :initform (symbol-name (gensym "C"))
    :documentation
    "The externalized name of this html-element, for use in URLs or
interhtml-element references in HTML or client-side code. Guaranteed
unique.")
   (element-internal-name
    :accessor element-internal-name
    :initarg :element-internal-name
    :documentation
    "The name used by this html-element's parent to refer to it.")
   (value
    :initarg :value
    :initform nil
    :documentation
    "application 'value' of this html-element, returned by IMHO public
object protocol")
   (session
    :initarg :session
    :initform nil)
   (parent
    :accessor element-parent
    :initarg :parent
```

```
    :initform nil)
   (children
    :initform (make-hash-table)
    :documentation
    "A hashtable of children that are dynamically rendered by this
html-element; keys are the internal names of these children."))
  (:documentation
   "Base display html-element for applications")
  )

;;
;; Hashtable of children
;;

(defmethod element-children ((element html-element))
  (slot-value element 'children))

;;
;; Children which should have values taken
;;

(defmethod element-active-children ((element html-element))
  (let (children)
    (maphash (lambda (k v)
       (declare (ignore k))
       (setq children (cons v children)))
     (element-children element))
    children))

(defgeneric html-element-all-children (t)
  (:documentation
   "Given a child, returns a list of all children composing that
child."))

(defmethod html-element-all-children ((element html-element))
  (let ((children (list element)))
    (maphash (lambda (k v)
       (declare (ignore k))
       (setq children (append (html-element-all-children v) children)))
     (element-children element))
    children))

;; ------------------------------------------------------------
;; clos-method: initialize-instance
;;
;; default initializer for html-elements.

(defmethod initialize-instance ((element html-element) &rest initargs)
;  (declare (ignore initargs))
  (call-next-method)
  (if (and (slot-boundp element 'ext-name)
   (not (slot-boundp element 'int-name)))
```

```
      (setf (slot-value element 'int-name)
    (slot-value element 'ext-name))))

;      (let ((name (symbol-name (gensym "C")))) ;; (symbol-name (type-of element))))))
;;;  (if (and (not (slot-boundp element 'session))
;    (element-parent element))
;      (setf (slot-value element 'session)
;     (slot-value (element-parent element) 'session))))

;; ---------------------------------------------------------------
;; framework-method: render-html
;;
;; This is the default renderer for html-elements. If this method has not
;; been specialized to a subclass, we look around for an html template
;; for this html-element.

(defmethod render-html ((element html-element) stream)
  (let ((template (html-template *active-application* (type-of element))))
    (dolist (item (html-template-content template))
      (ecase (car item)
(:string
 (format stream (cadr item)))
(:child
 (let ((child (child-element element (cadr item))))
   (if child
       (render-html child stream)
       (format stream "[missing child: '~a']" (cadr item)))))))))

(defmethod render-html :around ((element html-element) stream)
  (with-slots (ext-name int-name)
    element
    (let ((sname (symbol-name (type-of element))))
      (format stream "~%~%"
      sname int-name ext-name)
      ;; keep track of the classes of components on this page.
      (setf (gethash (type-of element) *active-components*) t)
      (call-next-method)
      (format stream "~%~%"
      sname int-name ext-name))))

;; ---------------------------------------------------------------
;; accessors for 'html-element value'

;; FIX: differentiate between display value and internal value? - JLB

(defgeneric element-value (t)
  (:documentation
   "This function provides a means of communicating
application-meaningful values in and out of HTML and client side
representations.  Values come from and go to the client via the pair
of functions 'get-values-from-response' and 'render-html', and are got
and set by server code via this function and its corresponding setf.")
```

```
  )

(defmethod element-value ((element html-element))
  (slot-value element 'value))

(defmethod set-element-value ((html-element html-element) value)
  (setf (slot-value html-element 'value) value))

(defsetf element-value set-element-value)

;; ------------------------------------------------------------
;; Build an URL for a html-element.

(defmethod element-url ((element html-element) &key (method nil) (arg nil))
  (let ((path (concatenate 'string *active-url* (slot-value element 'ext-name) "/")))
    (if method
(progn
  (setq path (concatenate 'string path method))
  (if arg
      ;; FIXME: do inverse url argument encoding
      (setq path (concatenate 'string path "?" arg)))))
    path))

(defmethod element-url (no-parent &key (method nil) (arg nil))
  (let ((path (concatenate 'string *active-url* "parentless" "/")))
    (if method
(progn
  (setq path (concatenate 'string path method))
  (if arg
      ;; FIXME: do inverse url argument encoding
      (setq path (concatenate 'string path "?" arg)))))
    path))

;; ------------------------------------------------------------
;; get/set a child element

(defmethod child-element ((element html-element) int-name)
  (or (gethash int-name (element-children element))
      (make-instance 'static-string :value (format nil "[Missing Child: ~a]" int-name))))

(defmethod set-child-element ((element html-element) int-name child)
  (setf (element-parent child) element
(slot-value child 'int-name) int-name
(gethash int-name (element-children element)) child)
  child)

(defsetf child-element (element int-name) (child)
  '(set-child-element ,element ,int-name ,child))

;; ------------------------------------------------------------
;; Set a bunch of child elements to instances.
;; Call like this;
```

```
;;
;;    (instantiate-children
;;         parent
;;        '((child-name make-instance-arg1 make-instance-arg2 ...) ...))

(defun instantiate-children (parent param-list)
  (flet ((instantiate
          (lst)
          (let ((child-name (car lst))
                (make-instance-args (cdr lst)))
            (setf (child-element parent child-name)
                  (apply #'make-instance make-instance-args)))))
    (mapc #'instantiate param-list)))

(defmacro child-value (ele child)
  `(element-value (child-element ,ele ,child)))

;; ---------------------------------------------------------------
;; Bind multiple child values from a html-element instance

(defmacro with-children (values element &body body)
  (let ((comp (gensym)))
    `(let ((,comp ,element))
       (declare (ignorable ,comp))
       ,@(let ((element element))
   (and (symbolp element)
                `((declare (variable-rebinding ,comp ,element)))))
       ,comp
       (symbol-macrolet ,(mapcar #'(lambda (value-entry)
     (let ((value-name
    (if (symbolp value-entry)
value-entry
(car value-entry)))
   (child-name
    (if (symbolp value-entry)
value-entry
(cadr value-entry))))
       `(,value-name
 (element-value (child-element ,comp ',child-name)))))
 values)
,@body))))

(defmethod make-html-element ((session http-session) element-class &rest initargs)
  (let* ((instance-args (append (list element-class :session session) initargs))
 (element (apply #'make-instance instance-args)))
    (setf (session-element session) element)
    element))

;; ---------------------------------------------------------------
;; framework class: html-form
;;
;; A form html-element
```

```
(defclass html-form (html-element)
  ((method
    :accessor method
    :initarg :method)
   (target
    :accessor form-target
    :initarg :target)
   (form-children
    :accessor form-children
    :initform nil
    :initarg :form-children))
  (:documentation
   "Provides a mechanism for managing interaction with an HTML
form. Children of an instance of html-form will automatically
have their values set and extracted.")
  )

;; ---------------------------------------------------------
;; clos-method: initialize-instance
;;
;; Establish the default target for this form

(defmethod initialize-instance ((form html-form) &rest initargs)
  (declare (ignore initargs))
  (call-next-method)
  (if (not (slot-boundp form 'target))
      (setf (slot-value form 'target)
    (or (element-parent form) form)))
  #+broken
  ;; todo: this was causing too many problems, but in the abstract
  ;; might still be a good idea.
  (if (not (slot-boundp form 'method))
      (error "No method specified for ~A~%" form))
  )

(defmethod take-values-from-request ((form html-form) request)
  (dolist (target (form-children form))
    (let ((value (cadr (assoc (car target)
      (request-http-client-content request)
      :test #'equal))))
      (funcall (cadr target) value)))
  t)

(defmethod render-html :around ((html-element html-form) stream)
  (with-slots (ext-name target method)
    html-element
    (let ((action (element-url (or target html-element) :method method)))
      (with-tag (:stream stream :tag "FORM" :attr `(("METHOD" . "POST")
    ("NAME" . ,ext-name)
    ("ACTION" . ,action)))
(call-next-method)))))
```

```
;; -------------------------------------------------------------
;; framework class: html-form-element

(defclass html-form-element (html-element)
  ((parent-name
    :accessor external-form-name
    :initform nil))
  )

(defmethod set-child-element ((form html-form)
      int-name
      (child html-form-element))
  (call-next-method)
;;  (format t ";; AC: ~s -> ~s~%" form child)
  (setf (external-form-name child) (element-external-name form))
  (setf (form-children form)
(cons (list (element-external-name child)
    (lambda (value)
      (setf (element-value child) value)))
      (form-children form))))

;; -------------------------------------------------------------
;; framework class: html-form-element

(defclass labelled ()
  ((label
    :accessor field-label
    :initarg :label
    :initform nil))
  )

(defmethod render-html :around ((labelled labelled) stream)
  (with-slots (label)
    labelled
    (if label
(with-tag (:stream stream :tag "TABLE" :attr '(("BORDER" . "0")
       ("CELLSPACING" . "0")
       ("CELLPADDING" . "0")))
  (with-tag (:stream stream :tag "TR")
    (with-tag (:stream stream :tag "TD" :attr '(("VALIGN" . "MIDDLE")))
      (write-string label stream)
      (write-string " " stream))
    (with-tag (:stream stream :tag "TD" :attr '(("VALIGN" . "BOTTOM")))
      (call-next-method))))
(call-next-method))))

;; -------------------------------------------------------------
;; html-element: popup-list

(defclass popup-list (html-form-element labelled)
  ((popup-values
```

```
    :accessor popup-values
    :initform nil
    :initarg :list-values
    :documentation
    "A function that returns an alist of strings and values for the html-element."))
  )

(defmethod render-html ((element popup-list) stream)
  (with-slots (ext-name)
    element
    (with-tag (:stream stream :tag "SELECT" :attr `(("SIZE" . "1")
    ("NAME" . ,ext-name)))
      (dolist (x (funcall (popup-values element)))
(with-tag (:stream stream :tag "OPTION" :attr `(("VALUE" . ,(car x))))
  (write-string (cdr x) stream))))))

;; -------------------------------------------------------------
;; html-element: submit-button

(defclass submit-button (html-form-element)
  ((display-string
    :accessor display-string
    :initform "Submit"
    :initarg :value))
  )

(defmethod render-html ((button submit-button) stream)
  (with-slots (ext-name display-string)
    button
    (with-tag (:stream stream :tag "INPUT" :noclose t
       :attr `(("TYPE" . "SUBMIT")
       ("NAME" . ,ext-name)
       ("VALUE" . ,display-string))))))

;; -------------------------------------------------------------
;; html-element: text-field

(defclass text-field (html-form-element labelled)
  ((visible
    :initform t)
   (columns
    :initarg :cols
    :initform 30))
  )

(defmethod element-value ((field text-field))
  (or (call-next-method) ""))

(defmethod render-html ((field text-field) stream)
  (with-slots (ext-name visible columns)
    field
    (let* ((value (element-value field))
```

```
           (string (typecase value
                     (function (funcall value))
                     (t ""))))
      (with-tag (:stream stream
                         :tag "INPUT"
                         :noclose t
                         :attr `(("TYPE" . ,(if visible "TEXT" "PASSWORD"))
                                 ("SIZE" . ,(format nil "~d" columns))
                                 ("NAME" . ,ext-name)
                                 ("VALUE" . ,string)))))))

;; ---------------------------------------------------------------
;; html-element: text-area

(defclass text-area (html-form-element labelled)
  ((wrap-type
    :initarg :wrap
    :initform :hard)
   (columns
    :initarg :cols
    :initform 40)
   (rows
    :initarg :rows
    :initform 5)
   )
  )

(defmethod text-area-wrap-attribute ((field text-area))
  (with-slots (wrap-type)
    field
    (case wrap-type
      (:hard
       "HARD")
      (t
       "HARD"))))

(defmethod element-value ((field text-area))
  (or (call-next-method) ""))

(defmethod render-html ((field text-area) stream)
  (with-slots (ext-name rows columns)
    field
    (let* ((value (element-value field))
           (string (typecase value
                     (function (funcall value))
                     (t ""))))
      (with-tag (:stream stream
                         :tag "TEXTAREA"
                         :attr `(("TYPE" . "TEXTAREA")
                                 ("WRAP" . ,(text-area-wrap-attribute field))
                                 ("ROWS" . ,(format nil "~d" rows))
                                 ("COLS" . ,(format nil "~d" columns))
```

```
                                  ("NAME" . ,ext-name)))
        (write-string string stream)))))

;; -------------------------------------------------------------
;; html-element: fancy-text-field

(defclass fancy-text-field (html-form-element labelled)
  ((visible
    :initform t))
  )

(defmethod element-value ((field fancy-text-field))
  (or (call-next-method) ""))

(defmethod render-html ((field fancy-text-field) stream)
  (with-slots (ext-name)
    field
    (with-tag (:stream stream :tag "INPUT" :noclose t
       :attr '(("TYPE" . "HIDDEN")
       ("NAME" . ,ext-name))))
    (with-tag (:stream stream :tag "APPLET"
       :attr '(("CODE" . "ValidTextInput.class")
       ("CODEBASE" . "http://cafe.onshore.com/lang/java/test")
       ("WIDTH" . "90") ("HEIGHT" . "35")
       ("NAME" . ,(concatenate 'string ext-name "A"))
       ("MAYSCRIPT")))
      (param-tag stream "form-target" (external-form-name field))
      (param-tag stream "hidden-target" ext-name))))

;; -------------------------------------------------------------
;; html-element: password-field

(defclass password-field (text-field)
  )

(defmethod initialize-instance ((field password-field) &rest initargs)
  (declare (ignore initargs))
  (call-next-method)
  (setf (slot-value field 'visible) nil))

;; -------------------------------------------------------------
;; html-element: checkbox

(defclass checkbox (html-form-element labelled)
  )

(defmethod set-element-value ((element checkbox) value)
  (format t ";; checkbox value : ~S~%" value)
  (setf (slot-value element 'value)
(equal value "on")))

(defmethod render-html ((element checkbox) stream)
```

```
  (with-slots (ext-name)
    element
    (with-tag (:stream stream
       :tag "INPUT"
       :attr `(("TYPE" . "CHECKBOX")
       ("NAME" . ,ext-name))))))

;; -------------------------------------------------------------
;; html-element: radio-button

(defclass radio-button (html-form-element labelled)
  ((group
    :initarg :group
    :initform "RADIO"
    :documentation
    "Group to which a radio button belongs")
   (checked
    :initarg :checked
    :initform nil
    :documentation
    "Whether this button is initially checked in its group"))
  )


;; A radio button returns t if it's on, nil otherwise
(defmethod element-value ((button radio-button))
  (let* ((gname          (slot-value button 'group))
         (self           (slot-value button 'ext-name))
         (received-value (cadr
                          (assoc
                           gname (request-http-client-content *active-request*)
                           :test #'string-equal))))
    (string-equal received-value self)))

(defmethod render-html ((element radio-button) stream)
  (with-slots (ext-name group checked)
    element
    (with-tag (:stream stream
       :tag "INPUT"
       :noclose t
       :attr `(("TYPE" . "RADIO")
       ("NAME" . ,group)
       ,@(if checked '(("CHECKED")))
       ("VALUE" . ,ext-name))))))

;; -------------------------------------------------------------
;; html-element: reset-button

(defclass reset-button (html-element)
  )

(defmethod render-html ((reset-button reset-button) stream)
```

```
  (format stream "<INPUT TYPE=RESET>"))

;; ---------------------------------------------------------------
;; html-element: file-chooser

(defclass file-chooser (html-element)
  )

(defmethod render-html ((file-chooser file-chooser) stream)
  (format stream "<INPUT TYPE=FILE>"))

;; ---------------------------------------------------------------
;; html-element: image-input

(defclass image-input (html-element)
  )

(defmethod render-html ((image-input image-input) stream)
  (format stream "<INPUT TYPE=IMAGE>"))

;; ---------------------------------------------------------------
;; html-element: button

(defclass button (html-element)
  ()
  (:documentation
   "An HTML form button.\\footnote{FIX - This one isn't implemented
yet.}")
  )

(defmethod render-html ((button button) stream)
  (format stream "<INPUT TYPE=BUTTON>"))



;; -*- Syntax: Ansi-Common-Lisp; Base: 10; Mode: lisp; Package: imho -*-
;; $Id: tutorial.sgml,v 1.9 2001/04/07 05:41:50 apharris Exp $

(in-package :imho)

;;
;; Basic non-form elements
;;

;; ---------------------------------------------------------------
;; html-element: html-page

(defclass html-page (html-element)
  )

(defmethod render-html :around ((page html-page) stream)
  (with-tag (:stream stream :tag "HTML")
```

```
    (with-tag (:stream stream :tag "HEAD")
      (with-tag (:stream stream :tag "TITLE")
(format stream "PAGE")))
    (with-tag (:stream stream :tag "BODY"
       :attr '(("BGCOLOR" . "#ffffff")))
      (call-next-method))))


;; ---------------------------------------------------------------
;; html-element: static-string

(defclass static-string (html-element)
  )

(defmethod render-html ((element static-string) stream)
  (let* ((value (element-value element))
 (string (typecase value
   (string (if (equal value "") " " value))
   (function (funcall value))
   (t " "))))
    (write-string string stream)))


;; ---------------------------------------------------------------
;; html-element: hyperlink
;;
;; FIX: support hyperlinked images

(defclass hyperlink (html-element)
  ((method
    :accessor method
    :initarg :method
    :initform nil)
   (reference
    :accessor reference
    :initarg :reference
    :initform nil))
  )

;; ---------------------------------------------------------------
;; framework-method: element-value
;;
;; Allow the 'value' slot of a hyperlink to be a zero-arity function
;; that returns the content for the link.

(defmethod component-value ((hyperlink hyperlink))
  (with-slots (value)
    hyperlink
    (etypecase value
      (function (funcall value))
      (string value))))
```

```
;; TODO: "precompile" the output text for the url (don't call format
;; too often inside of 'render'

(defmethod render-html ((hyperlink hyperlink) stream)
  (with-slots (method reference)
    hyperlink
    (let (href)
      (cond (method
     (setq href (element-url (element-parent hyperlink) :method method)))
            (reference
             (setq href (element-url (element-parent hyperlink) :method reference)))
;;      (setq href reference))
    (t
     (error "hyperlink without reference")))
      (with-tag (:stream stream :tag "A" :attr '(("HREF" . ,href)))
(format stream "~A" (element-value hyperlink))))))


;; ------------------------------------------------------------
;; html-element: table-interior

(defclass table-interior (html-element)
  ((elements
    :accessor elements
    :initarg :elements
    :initform nil
    :documentation
    ;; todo -- settle this question:
    "List of cells (strings, hyperlinks, or other html-elements?).")
   (rows
    :accessor rows
    :initarg :rows
    :initform nil
    :documentation
    "The number of rows in this table.")
   (columns
    :accessor columns
    :initarg :columns
    :initform nil
    :documentation
    "The number of columns in this table.")
   (align
    :accessor align
    :initarg :align
    :initform "center"
    :documentation
    "Horizontal alignment of the cell contents in this table.")
   (valign
    :accessor valign
    :initarg :valign
    :initform ""
    :documentation
```

```
    "Vertical alignment of the cell contents in this table.")
   (colspan
    :accessor colspan
    :initarg :colspan
    :initform "1"
    :documentation
    "Number of columns each row occupies."))
  (:documentation
   "The inside of an html table, that is, the rows and columns.
Does not include the table declaration itself."))


(defmethod render-html ((guts table-interior) stream)
  (let ((elements (elements guts)))
    (do ((row 0 (incf row)))
        ((or (null elements) (and (rows guts) (= row (rows guts)))))
      (with-tag (:stream stream :tag "TR")
        (do ((col 0 (incf col)))
            ((or (null elements) (and (columns guts) (= col (columns guts)))))
          (with-tag (:stream stream :tag "TD"
                            :attr `(("ALIGN"   . ,(align guts))
                                    ("COLSPAN" . ,(colspan guts))
                                    ("VALIGN"  . ,(valign guts))))
            (let ((element (car elements)))
              (typecase element
                (string       (format stream (if (equal element "")
                                                 " "
                                                 element)))
                (number       (format stream "~A" element))
                (html-element (render-html element stream))
                (function     (funcall element)) ;; heck, why not?
                (t            " "))
              (setq elements (cdr elements)))))))))
```

4.2. Important Variables

4.2.1. Request Dynamic Variables

The variables described here are bound for the duration of an active request, from the point that an active session is identified, to the close of the output stream to the client.

Table 4-1. Dynamic Variables

Variable Name	Meaning
active-application	The active application.
active-session	The active session object.

Variable Name	Meaning
active-request	The active request object.
active-url	The active (base) URL of the element being constructed.
active-components	A hash table of component classes that is constructed as the page/return elemnt is assembled.
response-type	The MIME type of the response.

4.2.2. Other Variables

These others mostly have to do with logging and error recovery behavior.

Table 4-2. IMHO Variables

Variable Name	Meaning
lisp-server-port	The port on which IMHO listens for incoming JServ requests. Default is 8007.
imho-serving	't' if requests are being accepted for processing.
production	A variable that currently serves to disable interactive debugger activation.

4.3. Functions

4.3.1. Session Instances

'Session Instances' provide a means of reusing objects within unique sessions.

```
(defun session-instance (class &rest initargs)
  (let ((instance (funcall #'ensure-session-instance class initargs)))
    (apply #'reinitialize-instance instance initargs)))
```

4.3.2. Rendering Elements

```
(defgeneric render-html (t t))
```

4.3.3. Methods

Methods are made available to clients using define-wm:

```
(defmacro define-wm (method-name method-ll &body method-body)
  `(labels ((wm-args (arglist)
     (values (mapcar #'car arglist)
     (mapcar #'cadr arglist)))
```

```
    ;; Should we install new objects into the session here?
    (wm-lambda (args &rest body)
     (compile nil (coerce '(lambda ,args ,@body) 'function))))
    (destructuring-bind (type &rest args)
',method-ll
      (let ((name (string-downcase (symbol-name ',method-name))))
(multiple-value-bind (arg-vars arg-types)
    (wm-args args)
  (let ((body-func (wm-lambda (cons (car type) arg-vars) '(progn ,@method-body))))
    (setf (gethash name *methods*)
  (make-wmethod :name name :type type
:arguments arg-types :body body-func))))))))


(defun process-wm-args (method-args request-args)
  (flet ((process (method-arg ext-arg)
   (case method-arg
     ('string
      ext-arg)
     (t
      (intern-ref method-arg ext-arg)))))
    (mapcar #'process method-args request-args)))

(defvar *methods* (make-hash-table :test #'equal))

(defun lookup-wm (name &optional default)
  (or (gethash name *methods*)
      (and default (gethash default *methods*))))

(defmacro undefine-wm (method)
  `(setf (gethash (string-downcase (symbol-name ,method)) *methods*) nil))

(defmacro refer-wm (method &rest args)
  `(let ((method (lookup-wm (string-downcase (symbol-name ',method))))
 (argv (list ,@args)))
    (format t ";; Args: ~s~%" (list ,@args))
    (if (not method)
(error "No reference for method ~s" ',method))
    (let ((name (wmethod-name method)))
      (if argv
  (progn
    (setf name (concatenate 'string *active-url* "0/" name "?"))
    (mapc (lambda (arg)
    (format t ";; Arg: ~s~%" arg)
    (setf name (concatenate 'string name (extern-ref arg)))
    )
  argv)))
      name)))
```

4.3.4. Passing arguments to Methods

In order to pass arguments to methods, ther must be a means of allowign the client to uniquely identify objects which exist on the server; this is done using the intern-ref/exter-ref pair of functions.

```
(defgeneric intern-ref (t t))

(defmethod intern-ref (object arg)
  (error "No default internalizer for objects of type '~s'" object))

(defgeneric extern-ref (t))

;; A string externalizes as itself, mutatis mutandis URI escaping.

(defmethod extern-ref ((string string))
  string)
```

4.3.5. Starting and stopping apps

The init/application function takes a keyword argument of :start, :stop, :restart, or :report. To start an IMHO application, type: * **`(init/application 'foo-app :start)`**

And to stop it: * **`(init/application 'foo-app :stop)`**

```
;; ------------------------------------------------------------
;; function passed to export-url

(defun init/application (app-class state)
  (let* ((app (make-instance app-class))
         (url (base-url app)))
    (ecase state
      (:start
       (unless (gethash url *imho-active-apps*)
         (setf (gethash url *imho-active-apps*) app)
         (application-startup app)
         (init/imho :start)))
      (:stop
       (when-bind (app (gethash url *imho-active-apps*))
         (application-shutdown app)
         (remhash url *imho-active-apps*)))
      (:restart
       (init/application app-class :stop)
       (init/application app-class :start))
      (:report
       (gethash url *imho-active-apps*))))
  (values))
```

4.3.6. Generating SGML markup

```
;; Functions for programmatically generating SGML markup
;;

;; with-tag - A macro that wraps some code inside an SGML tag.
;;
;; example:
;;
;; (with-tag (:tag "P")
;;   (with-tag (:tag "I")
;;     (format t "It was a dark and stormy night.")))
;;
;; expands to code that writes:
;;
;; <P><I>It was a dark and stormy night.</I></P>
;;
;; the body is optional, and if you want to omit the close tag, say:
;;
;; (with-tag (:tag "IMG" :noclose t :attr '(("SRC" . "http://localhost/gif.jpg"))))
;;
;; which produces
;;
;; <IMG SRC="http://localhost/gif.jpg">

(defmacro with-tag ((&key tag (stream '*standard-output*)
  (attr nil) (noclose nil)) &body body)
  `(locally
    (let ((stream ,stream))
      (format stream "<~A" ,tag)
      (if ,attr
  (dolist (pair ,attr)
    (if (null (cdr pair))
(progn
  (write-char #\  stream)
  (write-string (car pair) stream))
(format stream " ~A=\"~A\"" (car pair) (cdr pair)))))
      (format stream ">~%")
      ,@body
      (if (not ,noclose)
  (format stream "</~A>~%" ,tag)))))

(defmacro param-tag (stream name value)
  `(with-tag (:stream ,stream
      :tag "PARAM"
      :noclose t
      :attr `(("NAME" . ,name)
      ("VALUE" . ,value)))))

(defmacro with-reference ((reference stream) &body rest)
  `(with-tag (:stream ,stream
      :tag "A"
```

```
      :attr `(("HREF" . ,reference)))
    ,@rest))

;; Now, rather than
;;
;; (with-tag (:stream stream :tag "A" :attr `(("HREF" . (refer-wm frob widget))))
;;   (write-string "Frob the Widget" stream))
;;
;; you say
;;
;; (with-action (stream frob widget)
;;   (write-string "Frob the Widget" stream))

(defmacro with-action ((stream method &rest args) &body body)
  `(with-reference ((refer-wm ,method ,@args) ,stream)
    ,@body))

;; Someone make this work, wah!

(defun image-tag-attributes (url text size spacing border)
  (labels ((itoa (i)
     (format nil "~d" i)))
    (let ((atts (list (cons "ALTTEXT" text)
      (cons "SRC" url))))
      (if size
  (setq atts (cons (cons "WIDTH" (itoa (car size)))
   (cons (cons "HEIGHT" (itoa (cadr size)))
 atts))))
      (cons (cons "BORDER" (itoa border))
    (cons (cons "HSPACE" (itoa spacing))
  (cons (cons "VSPACE" (itoa spacing))
atts))))))

(defmacro image-tag (stream url text &key (size nil) (spacing 0) (border 0))
  `(with-tag (:stream ,stream :tag "IMG" :noclose t
      :attr (image-tag-attributes ,url ,text ,size ,spacing ,border))))
```

4.3.7. Logging

```
(defvar *imho-log-path* #p"/var/log/imho/*.log")

(defvar *imho-log-file* "imho")

(defvar *imho-log-stream* nil)

(defun log-event (event)
  (let ((log-stream (ensure-log))
        (remote-ip "Unknown Client"))
    (if *active-request*
```

```
        (setf remote-ip (request-http-client-ip *active-request*)))
    (format log-stream
            "~s - user - [~a] ~a~%"
            remote-ip
            (ext:format-universal-time nil (get-universal-time))
            event)
    (force-output log-stream)))
```

4.3.8. Communicating With Client-side Java

```
;; ============================================================
;; Stuff for writing Java types to a stream or a file
;;
;;

(defvar *string-svid* #xadd256e7e91d7b47)

(defvar *object-stream-magic* #xaced)
(defvar *object-stream-default-version* #x0005)

(defconstant +os-null+ #x70)
(defconstant +os-ref+ #x71)
(defconstant +os-class-desc+ #x72)
(defconstant +os-obj+ #x73)
(defconstant +os-string+ #x74)
(defconstant +os-array+ #x75)
(defconstant +os-class+ #x76)
(defconstant +os-blockdata+ #x77)
(defconstant +os-endblockdata+ #x78)
(defconstant +os-reset+ #x79)
(defconstant +os-blockdatalong+ #x7a)
(defconstant +os-exception+ #x7b)

(defclass object-output-stream ()
  ((handle
    :initform 0)
   (binary-stream
    :initarg :stream))
  )

(defun make-object-output-stream (stream)
  (let ((oos (make-instance 'object-output-stream :stream stream)))
    (write-object-stream-header oos)
    oos))

(defmethod write-object-stream-header ((stream object-output-stream))
  (with-slots (binary-stream)
    stream
    (write-int16 *object-stream-magic* binary-stream)
    (write-int16 *object-stream-default-version* binary-stream)))
```

```
(defgeneric write-java-object (t t)
  (:documentation
   "Write an object onto an object output stream, barf if you don't
know how to do it."))

(defmethod write-java-object ((type t) (stream object-output-stream))
  (error "Don't know how to serialize a ~s for Java" (type-of type)))

(defun write-java-utf (string stream)
  (let ((len (length string)))
    (write-int16 len stream)
    (do ((x 0 (incf x)))
((= x len))
      (write-byte (char-code (aref string x)) stream))))

(defmethod write-java-object ((string string) (stream object-output-stream))
  (with-slots (binary-stream)
    stream
    (write-byte +os-string+ binary-stream)
    (write-java-utf string binary-stream)))

(defun write-java-class-desc (classname binary-stream)
  (write-byte +os-class-desc+ binary-stream)
  (write-java-utf classname binary-stream)
  serialVersionUID
  newHandle
  classDescInfo)

(defmethod write-java-object ((string-array simple-vector) (stream object-output-stream))
  (with-slots (binary-stream)
    stream
    (let ((len (length string-array)))
      (write-byte +os-array+ binary-stream)
      (write-byte +os-class-desc+ binary-stream)
      (write-java-utf "[Ljava.lang.String;" binary-stream)
      (write-int64 *string-svid* binary-stream)
      (write-byte #x02 binary-stream)
      (write-int16 #x0000 binary-stream)
      (write-byte +os-endblockdata+ binary-stream)
      (write-byte +os-null+ binary-stream)

      (write-int len binary-stream)
      (do ((x 0 (incf x)))
  ((= x len))
(write-java-object (aref string-array x) stream)))))

(defmacro with-java-stream ((stream file) &rest body)
  `(with-open-file (bs ,file :direction :output :element-type 'unsigned-byte)
    (let ((,stream (make-object-output-stream bs)))
      (with-slots (binary-stream)
,stream
```

```
,@body))))

(defun test ()
  (with-java-stream (stream "/tmp/string-array-l.bin")
    (write-java-object #("Uno" "Ena whena booda poo" "The REPL Who Loved Me" "From Symboli
;;    (write-java-object "Live and Let Cons" stream)
    ))
```