DYALOG

The tool of thought for expert programming

Dyalog for Windows

MiServer User Guide

Version 2.0

Dyalog Limited

Minchens Court
Minchens Lane
Bramley
Hampshire
RG26 5BH
United Kingdom

tel: +44 (0)1256 830030
fax: +44 (0)1256 830031

email: support@dyalog.com
http://www.dyalog.com

For MiServer Version 2.0

First Edition February 2012

Contents

CHAPTER 1

An APL Window to the Internet

1.1 Web Server Fundamentals

Since you're reading this document, it's probably safe to assume that you've used a web browser to access web pages before. You probably know that a web page is a resource residing somewhere on a computer. When you enter a web address (URL) into your browser's address field, your browser sends a request for that resource to the computer on which the resource should be stored. A process on that computer, called a web server, listens for requests, interprets them, tries to locate the requested resource and finally returns it to the browser.

More specifically, a web server is a process that listens to a TCP/IP port for incoming connection requests. Once a connection has been established between the web server and a client, typically a web browser, communication begins between them according to a set of standards called the Hypertext Transfer Protocol (HTTP). When the server receives a request for a resource, it performs a number of preparations on that resource for transmission, responds back to the browser, and closes the connection. If all goes well, the browser displays a web page.

Today's servers often have a number of other useful capabilities. Servers can track individual clients through the use of sessions and cookies, saving data between requests. This supports features like shopping carts and multi-part forms. Many servers also support interactions with databases, user authentication, and resource specific access restrictions.

1.2 Introducing MiServer

MiServer is an open source web server implemented in Dyalog APL which brings the power of APL to the web. It contains both web server components and web content development utilities designed to facilitate what has become the project's credo. "Anyone who can write an APL function should be able to turn it into hosted web content, without using any components that are not included in a standard installation of Dyalog APL." MiServer's users can develop web content without having to learn much about the underlying nuances of web page implementation. However, the full breadth of HTML, JavaScript and related technologies remain at the developer's disposal.

MiServer began with Stefano Lanzavecchia's "WildServer," a more complex and more object-oriented web server than what MiServer was to become. Morten Kromberg took WildServer and "watered it down," making it easier for mere mortals to use. This was the "MildServer." Most recently APL Tools Group was charged with extending and enhancing the project. They added features and interfaces, transforming it into the MiServer.

One note, the term “MiServer” is used in a number of contexts, including:

- The MiServer Project, an open source initiative involving MiServer (for more information see Appendix H)
- A MiServer Skin, which overrides some of the base functionality of the MildServer class

1.3 Why MiServer?

There are a number of circumstances where you may want to use MiServer.

- APL is your preferred development environment
- You want to host your existing APL functionality on a web page
- You want to use a web browser as a user interface
- You want to merge APL with the array of tools available to web developers
- You want to integrate almost any Dyalog APL utility with your web content. Including:
    - SQAPL: which provides access to any ODBC-compliant database
    - SAWS: which consumes and provides web services
    - SharpPlot: a business graphics package included with Dyalog APL

MiServer is a stable web content delivery environment. We expect to add new features, both internally and in collaboration with the APL community. However, a more “industrial” web server such as Microsoft IIS, Apache, or IBM WebSphere may have features that are currently not present in MiServer. You may opt to use one of these servers in one of these following situations:

- You need to use a third party technology like Adobe ColdFusion
- You need to use technology that integrates with a specific web server
- You desire to also host other server protocols, FTP for instance. Conga, Dyalog’s TCP/IP communications tool, can be used to develop this capability, but extensions that allow for this have not yet been integrated with MiServer

An alternative for delivering APL functionality via the web is through ASP.NET as described in the Dyalog .NET Interface Guide. Finally, it should be noted that it is possible to use the MiServer to deliver content which passes through a commercial web server framework like IIS or Apache, allowing you to combine the lightweight flexibility of APL with management and security features of the commercial web frameworks.

CHAPTER 2

MiServer Architecture

2.1 Web Servers and Sites and Pages, Oh My!

Browsers request resources, which can be almost any file type. If this resource is rendered as a user interface in the browser, the resource is called a web page. When you type a URL into a browser's address bar, you are issuing a request to access a resource on a computer somewhere. A URL, like `http://www.dyalog.com`, references the root of a directory of a collection of such resources, called a website. A website can be as simple as a single file in a folder or can be an expansive collection of content and functionality to process and format that content. As discussed in Chapter 1, a web server, like MiServer, hosts websites and makes them available for requests.

MiServer Core Functionality

MiServer 's architecture is implemented across a number of files and requires a specific directory structure, described in section 2.3.

The files representing the core functionality are found in the `ServerRoot`[1]`/Core/` directory. In general, it should not be necessary to modify any of these files, which include:

- The MildServer class – this class implements the core functionality of the server
- The HTTPRequest class – this class parses the HTTP request and generates the HTTP response
- The Boot namespace – this namespace contains functionality to start and stop a MiServer
- The MildPage class – this serves as the base class for all MiPages

MiSites

A MiSite is a directory with a specific structure that MiServer can host as a web site. This directory, outlined in section 2.3, must include several specific scripted classes and XML documents which MiServer requires for minimal operation. Additionally, any other files can be contained in the directory, so long as they do not cause name conflicts with the required files. A sample MiSite, Demo, has been included with MiServer.

---

1 `ServerRoot` is used to refer to the directory where you have installed MiServer. Most examples in this document use `C:\MiServer\` as the `ServerRoot`.

MiServer Skins

The MildServer class implements basic server functionality but you may wish to implement additional behaviors. MiServer makes this easy to implement by exploiting its object oriented nature. The MildServer class contains several overridable methods which can be overridden in a class derived from MildServer. This scripted class, called a MiServer Skin, can implement behavior specific to your MiSite, such as usage logging, idle activities and error handling.

MiPages

A MiPage is a scripted class used to create a web page. When requested by a browser, a MiPage runs APL code to generate content, usually Hypertext Markup Language (HTML), which is sent back to the requesting browser.

MiPage Templates

Each MiPage can generate a complete HTML document. However, you may find that some or all of your web pages require common HTML or scripts, especially if you want to give your MiSite a consistent look and feel. A MiPage template is a class, derived from the MildPage base class which can additionally process the HTML generated by a MiPage. You can have any number of these templates in your site and can switch between them by changing the template from which a MiPage is derived.

2.2 What you'll need to know to build MiSites

While we would like you to be able to put websites together with only your knowledge of APL, there are some additional concepts which you will need to be familiar with, including:

- A very basic understanding Object Oriented programming concepts
- A basic understanding of Hypertext Markup Language (HTML)
- How to manage and edit APL scripted files

A Bit of OO

Object oriented (OO) programming is a programming paradigm centered on structures called objects. Objects are independent instances of a class, an object blueprint which describes a set of related functions and/or data. Each object can contain unique data.

A class may derive from another class, referred to as a base class. The derived class acquires the methods, fields and properties of its base class on top of its own. In OO speak, this is called inheritance. A base class may specify a method as overridable, which means that a derived class can define its own behavior for a method of the same name.

The elements of OO programming will not be unfamiliar to an APLer, but use different terminology. Functions are called methods. Variables are called either fields or properties. There are additional attributes that are specific to the OO versions of these elements not described here.

For more information on OO, check out the Introduction to OO in the Language Help of Dyalog APL and Introduction to Object Oriented Programming for APL Programmers found in the documentation supplied with Dyalog APL or at http://docs.dyalog.com.

An HTML Survival Guide

Hypertext Markup Language (HTML) is the fundamental building block of web content. It 'marks up' a document with identifiers that tell browsers how to treat each piece of content. The identifying marks, called tags, are words surrounded by angle brackets (`< >`). Most tags work in pairs, surrounding content with an opening and closing tag.

```
Opening tag → <span> Content affected by span tags.
                </span> ← Closing tag
```

Any content contained within tags are affected by it. In the below example, the `<i>` tag identifies its contained text as to be italicized.

```
       <i>The Quick Brown Fox</i>

The Quick Brown Fox

      The <i>Quick</i> Brown Fox

The Quick Brown Fox
```

Tags often affect the content of all other tags inside themselves, although they do not always pass on all of their attributes.

```
      <i>The <u>Quick</u> Brown Fox</i>

The Quick Brown Fox
```

Here are a few tags that you might find useful.

Example Tags	Description
`<html> </html>`	These must wrap all HTML documents
`<head> </head>`	Wraps around the header section of the document that deals with preparations for page load and resource loading
`<body> </body>`	Wraps all HTML that is marked for visual representation on the page
` `	Inserts a line break
`<a href="http://www.dyalog.com"> Dyalog</a>`	Creates a hyperlink with the text surrounded by the tags
`<div> </div>`	A tag that has no meaning on its own, a division allows you to specify a space in the document
`<img src="/images/dragon2011.png" />`	Displays an image
`<ul> <li>Item1</li> <li>Item2</li> </ul>`	An unordered list and its list items

Most HTML documents follow a similar structure, outlined below.

```
<html>
   <head>
The head tags contain formatting information that affects the page
       but is not directly visible. This can be used to reference
       style sheets, set title tags and meta-information, and much
       more.
   </head>
   <body>
The body tags contain content that can be visible to the user. Most
       of the tags used in this section affect, categorize or
       contain the visible representation, but there are cases
       where this is not true.
   </body>
</html>
```

An opening HTML tag can also contain a number of ‘attributes.’ These attributes convey additional information about the tag. Some attributes like 'id' or 'class' are common to all tags, while others like 'href' may be specific to a single tag. Attributes are specified with the following syntax:

```
<tag attributename="attributevalue">Contents</tag>
```

A few useful attributes:

Attributes	Description
`id`	An identifier for a unique HTML element that allows the element to be selected by other technologies (CSS, JavaScript, etc…)
`class`	An identifier that associates an HTML element with a group of elements, allowing that element to be selected by other technologies as part of that group.
`style`	Inserts styles
`href`	Specifies the link destination of an `<a>` tag

Comprehensive information about tags, attributes and HTML best practices can be found at the W3 Schools at http://www.w3schools.com.

XHTML

There is a huge amount of slightly "incorrect" HTML in web pages which is still used today (for example, elements with opening tags but no corresponding closing tag). Browsers have been extremely tolerant, doing their best to render bad HTML “reasonably” but often differently from one browser to the next. This code is often difficult to read, and difficult for programs to parse.

To deal with these issues, HTML 4.01 was blended with Extensible Markup Language, or XML, to create Extensible Hypertext Markup Language, or XHTML. This set of more stringent standards is designed to standardize the writing of HTML and allow it to be parsed by existing XML parsing programs.

For HTML to be considered XHTML its elements must:

- be properly nested
- always be closed
- be in lowercase
- have one root element

If at all possible, the use of XHTML is highly recommended. To assist you in this, all the HTML generating utilities packaged with MiServer should conform to the XHTML standards.

Scripted files

All of the classes and namespaces that compose MiServer, save the workspace itself, are kept on UTF-8 encoded files with a `.dyalog` extension. This allows code to be modular and easy to edit. It provides a mechanism for APL users to develop and share code, which aligns nicely with the goal of MiServer as an open source project. Also, any of these scripted files can be edited both from inside an APL session and from a text editor.

2.3 Directory Structures

MiServer

This is the basic directory structure required for MiServer. The files are divided into several sections based on their function, specifically:

- Core – the required components of the server
- Extensions – APL Extensions to server functionality
- Documentation
- Error Pages – html pages which can serve as responses
- Utils – Utility namespaces loaded on server start
- Plugins – Third party extensions

An installation of MiServer also includes a demonstration MiSite (Demo) which is not included in the following chart, as it is not a required part of the server directory structure.

Also not included is a complete listing of the contents of the Plugins directory.

Directory or File	Description
C:\MiServer\	The root directory of MiServer (SiteRoot) – can be any folder
mserver.dws	The MiServer workspace
Core\	Contains the core components of MiServer
Boot.dyalog	Namespace containing functions to start and stop MiServer
HTTPRequest.dyalog	Class which encapsulates all information for an HTTP request
MiPage.dyalog	Class which serves as a base class for all MiPages
MildServer.dyalog	Class which implements the MiServer core and serves as a base class for all MiServer Skins.
ContentTypes.xml	Configuration file containing the associations between file extension and HTTP `contenttype` header value
Extensions\	Folder for extensions to MiServer to implement additional functionality
SimpleAuth.dyalog	Implements basic HTTP authentication
SimpleSessions.dyalog	Implements stateful interaction using sessions
ContentEncoder.dyalog	An interface for implementing content encoding schemes, like HTTP compression
deflate.dyalog	Implements the default compression style
Lumberjack.dyalog	Logs HTTP requests
Documentation\	Documentation associated with the MiServer
MiServer User Guide.pdf	This document
Included Plugins.pdf	A list of the third party plugins distributed with MiServer
ErrorPages\	The pages sent to the browser to display information about errors
PlugIns\	Third party plug ins
JQuery\	Files associated with the JQuery JavaScript library
Utils\	Utility classes and namespaces
Base64.dyalog	Functions for encoding and decoding messages in base 64
Dates.dyalog	Functions dealing with dates
DrA.dyalog	Error logging functions
Files.dyalog	Functions to manipulate files
HTML.dyalog	Functions to assist in the creation of HTML
HTMLInput.dyalog	Functions to assist in the creation of HTML, focused on form and input objects
JQ.dyalog	Functions that integrate your MiPages with the JQuery JavaScript library
JQUI.dyalog	Functions that integrate your MiPages with the JQueryUI JavaScript library
JQO.dyalog	Functions which integrate your MiPages with third party JQuery scripts
SQL.dyalog	Functions that integrate your MiPages with the database interactions of SQAPL
SMTPMail.dyalog	For sending mail messages via SMTP

A Sample MiSite

A MiSite has a required directory structure, outlined below. While the exclusion of some of the below files may not crash MiServer, they should be considered the required as well.

Directory or File	Description
`C:\MiServer\Demo\`	The root of the Demo site. A MiSite can be installed in any directory and need not necessarily be installed in a directory under MiServer
`Admin\`	Contains `.dyalog` files that control configuration settings
`Code\`	Contains code to implement behavior specific to this MiSite. This folder will contain MiServer Skins should you choose to implement them
`Templates\`	Contains all MiPage templates should you choose to implement them
`Config\`	Contains XML files used for site specific configuration
`DrA\`	Contains error logs
`Scripts\`	Contains user created and third party scripts
`Styles\`	Contains cascading stylesheets and other resources that affect the look of your website
`error.css`	Cascading style sheet for error pages generated by the DrA error handling utility
`TempFiles\`	Contains temporary MiServer files
`Index.dyalog`	The default page that is loaded if no page is otherwise specified

CHAPTER 3

Getting Started

3.1 Installing and Running MiServer

Prerequisites

MiServer requires:

- The Windows or Linux edition of Dyalog APL version 12.1 or later
- At least version 2.2 of Conga, Dyalog's TCP/IP communications tool

Installation

The supported installation of MiServer should be available at `http://tools.dyalog.com`. Download the files and unzip them into any directory. In all following demonstrations, MiServer is assumed to have been installed in `C:\MiServer` (represented by `SiteRoot`).

3.2 Configuration

Server Configuration

When a MiServer is booted, it must at a minimum find the site configuration file `server.xml` in the directory `SiteRoot\Config\`. This configuration file is used to fill generate variables in `#.Boot.ms.Config`, which are referenced in various places in the MiServer architecture.

MiServer only references `Server.xml` at start up, so any changes to the server configuration will not be applied until the server is restarted. The xml file is text, so edits can be made to an XML file from most text editors. Additionally, you may add your own tags and values to this list to generate custom configuration settings that will be accessible by your MiPages.

Server.xml Parameters

Note: This table only gives an overview of each of the parameters. For a detailed discussion of each parameter, please see Appendix G.

Parameter	Example	Explanation
Name	MiServer Demo	The name of the MiSite
ClassName	DemoServer	Valid: MildServer or the class name of a MiServer Skin which is in SiteRoot/Code/
Lang	en	The language encoding of the majority of content on the site. This is primarily used by websites to determine dictionary and voice settings
Port	8080	The port on which the server will listen for incoming connections NOTE: Port 80 is the default port number used by HTTP servers. If you don't already have a web server installed you might want to use 80 to avoid having to specify a port number when browsing the site. This may require additional permissions
SessionHandler	SimpleSessions	The name of the class which will handle sessions
Authentication	SimpleAuth	The name of the class which will handle authentication
Logger	LumberJack	The name of the class which will handle server logging
UseContentEncoding	1	1 or 0 - use the content encodings specified in SupportedEncodings
SupportedEncodings	deflate	The names of the classes that use the ContentEncoder interface, separated by commas in the order of usage preference
LogMessageLevel	1	Determines which log messages are displayed. See Section 3.4
DefaultPage	index.dyalog	The file name of the resource to return if no page name is given by the browser
HttpCacheTime	2	The server can send information with your content that cues browsers to cache resources and refer to them locally Valid: 0=Off or the length of time (in minutes) a resource should be cached Note: This setting can cause issues during development if resources are frequently changed
IdleTimeOut	0	Valid: 0=Off or a number (in minutes) Amount of time before the server triggers idle state behaviors
UserID		UNIX Only – User ID to switch to after MiServer allocates a port
TrapErrors	0	Valid: 1=Trap and Log errors, 0=Crash
Debug	2	Valid: 0=No Debug Info, 1=Debug Info, 2=Allow Editing
MailMethod	NONE	Valid: SMTP,NET,NONE
MailRecipient		

Other Configuration Files

Depending on the extensions, plugins or utilities being used by your MiSite, MiServer may require additional configuration files to host a MiSite. Depending on the nature of the code using the code requiring the file, there could be any number of consequences related to not including a file.

The Demo MiSite includes several more, which configure a few extensions advanced options:

- Access.xml (See Section 8.6)
- Users.xml (See Section 8.6)
- Datasources.xml (See Chapter 7)
- Resources.xml (See Appendix F)
- Virtual.xml (See Appendix F)

3.4 MiServer Basics

Load the MiServer workspace.

```
      )load C:\MiServer\mserver.dws

C:\MiServer\mserver saved Tue Sep 20 12:07:45 2011
      Start 'Demo' ⍝ Run the demo
```

There are three functions in the root of the workspace which provide basic control over MiServer:

- #.Start – Starts a MiServer
  - takes a MiSite directory path as the right argument, as a full path or a path relative to the `ServerRoot`

```
      Start 'Demo'
MiServer started on port: 8080
```

- #.Stop – Shuts off the currently running MiServer

```
      Stop
MiServer stopped.
```

- #.Restart – Stops and starts the currently running MiServer

```
      Restart
MildServer stopped.
MiServer started on port: 8080
```

Start the Demo. Since it is configured with the default `Server.xml` file, it listens on port 8080 for HTTP requests. The server will fail if another program is using its configured port. If you have a firewall installed you may need to grant Dyalog APL internet access.

When the server has booted, open your web browser of choice and enter `http://localhost:8080`. This directs the browser toward your own computer, targeting port 8080.

You should see the following page:

Take a few minutes and click through the links on the bottom half of the index page. They lead to other MiPages which demonstrate various MiServer capabilities.

When you are done, run #.Stop.

3.5 Creating a New MiSite

The easiest way to create a new MiSite is to take an existing one and modifying it to suit your needs. Copy the entire `ServerRoot\Demo` directory to another directory on your computer, renaming the root folder as you choose.

Anytime we reference or make changes to a MiSite from now on, we will assume it to be in `C:\MyMiSite`.

Initialize your New MiSite

Start a MiServer on this directory. First, load MiServer:

```
      )Load C:\MiServer/mserver.dws
C:\MiServer\mserver saved Mon Sep 12 16:07:54 2011
      Start 'Demo' ⍝ Run the demo
```

Then use #.Start to initialize your MiSite:

```
      Start 'C:\MyMiSite'
MiServer started on port: 8080
```

CHAPTER 4

Your First MiPages

4.1 MiPages

MiPages are the building blocks of a MiSite. They are the resources that will use APL to make content. When requested by a browser, a MiPage is typically responsible for putting together data, which is usually HTML. It then hands that data off to MiServer to be sent to the browser.

Requirements of a MiPage

A MiPage is an APL class contained in a `.dyalog` scripted file. The class must:

- be specified as derived from the base class MildPage or a MiPage template
- contain a method named Render which must be:
  - public
  - monadic
  - passed the request object
  - and is required to pass a character string containing valid HTML to the function req.Return

4.2 Building Your First MiPage

Building a basic page

When requested, this first MiPage displays everyone's favourite test message, "Hello World". In the example below, we will build this MiPage from within the editor and save it as a `.dyalog` file in the root directory of the site.

In the session, type:

```
)ed helloworld
```

In the editor window, enter this text:

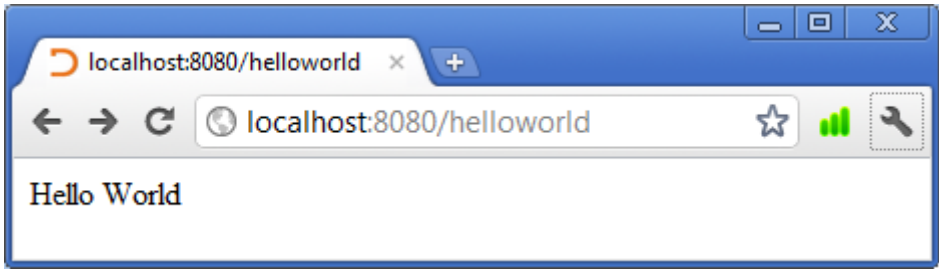
```
:Class helloworld : MildPage
```

```
Render req
   :Access Public
   req.Return 'Hello World!'



:EndClass
```

Save the class as a scripted file using the SALT utility:

```
]Save helloworld C:\MyMiSite/
```

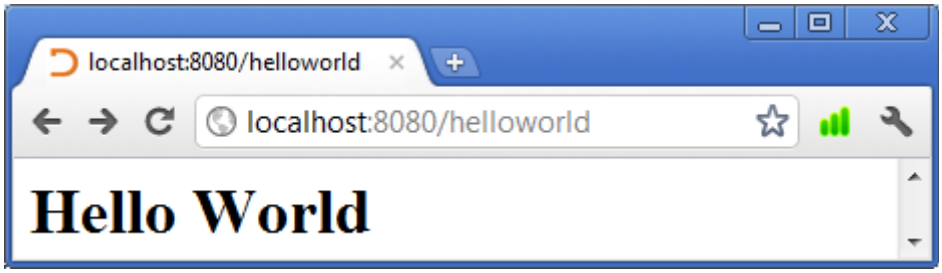
Making sure the MyMiSite server is running, open a web browser and type `http://localhost:8080/helloworld` into your address bar. You should see something like this:



You can see that the character vector found its way to the browser. Any simple character vector passed to `req.Return` will be returned to the browser and interpreted as HTML. That said, the MiPage could instead produce a message wrapped in headline tags:

```
:Class helloworld : MildPage


Render req
   :Access Public
   req.Return '<h1>Hello World!</h1>'



:EndClass
```

The above displays as:



4.3 Introduction to Templates

The previous examples return HTML but do not look much like a web site. We already have seen the demo site, which does looks quite a bit more like one. To use the styling of the rest of the demo, take the previous example and set its base class to MiPage.

```
:Class helloworld : MiPage
```

The demo site includes a template called MiPage.dyalog in its Code/Templates directory. It contains a class which additionally processes the HTML passed to req.Return and finishes building the HTML that will be sent to browsers. When we changed the base class, we applied that template to the code, which will make the page look like the following when requested:

If you select a template that does not exist or is not working properly, the MiPage will not be displayed.

4.4 Getting Dynamic

MiPages are APL code, and can consume and manipulate data. This data can come from anywhere APL can access, most pertinently from the client's browser. We will get into the nuts and bolts of that interaction in a few chapters, right now we are going to get the need to know answers for how to build MiPages that talk with browsers.

Note: You should familiarize yourself with the HTMLInput namespace. The MiPage below uses it for communicating between the server and the browser. Appendix A has a function reference for this namespace.

Reverse.dyalog

This first example of a 'dynamic' MiPage will be simple and APL idiomatic. There will be a field to enter a word. If there are characters in the field and the button underneath it is pressed, the page will refresh and the field will contain the characters in reverse order.

This MiPage already exists in the demo site. Since MyMiSite is a copy of that demo, `reverse.dyalog` is in the `SiteRoot`[2] directory.

The MiPage has been copied in below:

Reverse.dyalog

```
:Class Reverse : MiPage

:Include #.HTMLInput                ⍝ Useful functions for creating
                                    ⍝   HTML pages

:Field Public Text←''               ⍝ Name of edit field
:Field Public Action←''             ⍝ All action buttons have this name

    ∇ Render req;html
     :Access Public

     DoAction                       ⍝ If a button was pressed, deal
                                    ⍝ with it

     html←'<br />Enter Text: '
     html,←'Text'Edit Text          ⍝ An "Edit" called "Text"
                                    ⍝ containing the Text
     html,←'<br /><br />'
     html,←'Action'Submit'Reverse'  ⍝ A button named 'Action'
                                    ⍝  with Caption 'Reverse'
     html,←'Action'Submit'Clear'    ⍝ ... another button named 'Action'

     html←req('post'Form)html       ⍝ Put a 'submit' form around it

     html,←'a href="/"'Enclose'Home' ⍝ A link back to the index page

     req.Return html
    ∇

    ∇ DoAction
      :Select Action
      :Case 'Clear' ⋄ Text←''
      :Case 'Reverse' ⋄ Text←⌽Text
      :EndSelect
    ∇
```

2 `SiteRoot` is used to refer to the directory where you have installed your MiSite. Most examples in this document use `C:\MyMiSite\` as the `SiteRoot`

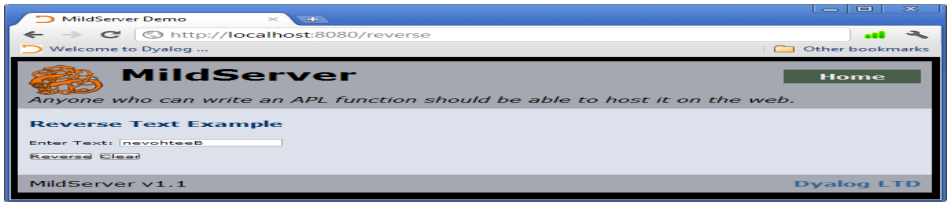
```
:EndClass
```

Reverse is derived from the MiPage template., containing two methods, Render and DoAction.

- Render generates the page's HTML
- DoAction:
  - handles the data passed from the client to the browser
  - selects an action depending on the button pushed

Put `http://localhost:8080/reverse?Name=Beethoven` into the URL line of your browser:

Click the "Reverse" button:



Getting Information from the Browser

If you enter text into the text field and click the 'Reverse' button, the value changes as we might expect it to. The text reverses. If you pay careful attention, you will notice that the page reloaded after you clicked the button. As you may have figured out from looking at the class, the value of the text type input element is passed to the Text field. The Render method is called and those values change the way the HTML is constructed.

Remember that HTMLInput contains functions that generate HTML, a number of which are associated with transferring data between the browser and the server. For all those that generate input elements, you can pass the name of a public field as the left argument. When the data is returned by a form submission (described immediately following this), it automatically fills that field with the value of the input element.

```
    'Text' #.HTMLInput.Edit Text
<input  type="text" size="" id="Text" name="Text" value="VALUEOFText" />
```

Forms: Quick and Dirty

A little more structure is required to get client data back to the server. You will need to wrap your input elements in a form, which is described in detail in Chapter 6. Right now, know that input elements on a page must be contained within a `<form>` tag to be 'valid' to have their values sent to the server. HTMLInput.Form wraps HTML within such a tag.

If you are not doing anything particularly tricky and can receive the values of all your HTML elements that pass value, simply to pass the variable that contains your HTML through HTMLInput.Form immediately prior to passing it to req.Return. Use the syntax below (notice that Form is a user defined operator):

```
req.Return req('post'Form)CharacterVectorOfYourHTML
```

Once the page is loaded, the client must trigger a new request that contains the form data, called a form submission. The aptly named 'submit button' is a basic way to do that. There are two different submit buttons on the example page, each generated with HTMLInput.Submit, which takes the name attribute as a left argument and the value attribute as the right argument. The value that is passed to the server during submission is the text on the button.

Because only the value of the pressed submit button is sent to the server, it is a common practice to give each submit button the same name and use a :Select test to determine which was pressed.

Preserving Data Past the Pageload

You may have noticed that while you can communicate with the server, the fields that may populate your pages are reset on every load. The Reverse example operates like this. The word passed to the server is not stored, but used to generate the new page.

You can also save data by defining a variable in the req.Session.State namespace. As described in Section 8.5, MiServer can identify requests from the same source and makes a unique copy of the namespace available to each user.

However, once the session ends, this data will be lost.

```
req.Session.State.ProductNumbers ← '1023' '0012' '3104'
```

4.4 Going Further

Now that you understand how to communicate between a MiPage and a browser, you can use your knowledge of APL and your growing understanding of HTML to build a wide array of content. What follows are specific concerns you will have when implementing your MiSite, instructions for server features, utilities you can use to build more interesting MiPages and an increasingly in-depth look at building web content.

CHAPTER 5

Bringing the Pieces Together – The MiSite

5.1 Planning Your Site

At its heart, a MiSite is a directory structure with a few key files. It must include:

- the directories listed in the chart in section 2.3
- a properly configured Server.xml file in the `Config` directory
- a default page in the root directory, properly named as per the configuration setting `DefaultPage`

With these requirements met, MiServer can run and serve the MiSite.

There are extensions to MiServer and utilities which require their own configuration on the site level. All of these must be found in the `Config` directory. Currently, those include:

- `Access.xml` – used by SimpleAuth (See Section 8.6)
- `Users.xml` – also used by SimpleAuth (See Section 8.6)
- `Databases.xml` – used by SQA (See Chapter 7)
- `Resources.xml` – used by MiServer to create resource mappings (See Appendix F)
- `Virtual.xml` – used by MiServer to define virtual resource (See Appendix F)

Also there are other files which MiServer will only be able to use if there are in certain places:

- MiServer Skins must be found in `SiteRoot/Code/`
- MiSite Templates must be found in `SiteRoot/Code/Templates/`

So where do all my MiPages go?

Any other files or directories you would like to add to the MiSite will not interfere with MiServer. However, if you are using an extension that adds user authorization requirements to some or all of your site, like SimpleAuth, clients may not be able to access those files without proper credentials.

5.2 The Default Page

Most frequently, when first surfing to a website, users do not request a specific resource but instead request the root directory of the site. When the root is requested, web servers look to see if there is a default page specified. You have already experienced this with MiServer when serving the demo

MiSite and navigating to `http://localhost:8080`. The resource that comes up in your browser is actually a file named `index.dyalog`.

The `DefaultPage` parameter of `Server.xml` contains the filename which MiServer will attempt to serve any time the root directory or a subdirectory is requested. If that resource is not present, MiServer will return a 404 - File Not Found error page.

This setting affects the default page name for every directory. If you plan on having multiple directories of resources which can be accessed my navigating to the path, each default page will need to be named the same.

5.3 Error Pages

When a web site cannot return a resource, its server customarily returns an error page to the client. Like the dreaded 404 page, there are several common status codes which communicate the state of the response and the types of errors. A list of these codes can be found at the W3 schools website: `http://www.w3schools.com/tags/ref_httpmessages.asp`

At several points throughout the MiServer architecture, errors are trapped and the status code is passed to `req.Fail`. MiServer will send error pages to client for errors in for following four categories:

- 401 – Unauthorized Access
- 404 – File Not Found
- 500 – Internal Server Error
- 501 – Header values specify a method that is not implemented

If `SiteRoot/ErrorPages` contains an HTML page named after one of these errors (i.e. 404.html) that file will served as the response when its corresponding error is triggered.

CHAPTER 6

Utilities for Building MiPage Content

6.1 The Utilities Library

One of the goals of the MiServer project is to provide users with tools to build web pages using APL like syntax. There are several To facilitate this, several utility namespaces are included which cover things that are commonly used to build web pages.

When the Start or Load commands are used, any Dyalog scripted file in the /Utils/ folder of the server directory is loaded into the root of the workspace. This means that users can develop their own tools and easily load them.

These include:

HTML	HTMLInput	JQ
JQO	JQM	JSON
SQL	JQUI	XML
Base64		

6.2 Utilities

General Notes

The following entries are descriptions of the utility files included with MiServer. Unless otherwise noted, the following utilities are cross platform.

HTML Utilities

HTML

This namespace contains simple cover functions for building HTML tags which wrap around character strings. Since it is based on a simple common structure, it is easily extensible. For more information, see APPENDIX A.

HTMLInput

HTMLInput was originally designed to build HTML forms. It has since expanded to become our most sophisticated HTML building toolset, and is relied on heavily for many of the examples you see in this manual.

JQuery Utilities

JQ

JQ is a namespace which contains basic functions for building JQuery scripts in a MiPage, as well as utility functions for building other JQuery covers.

JQUI

JQueryUI is an extension to the JQuery library which builds user interface 'widgets,' such as calendars, dialog boxes and tabbed content containers. JQUI provides basic access to these widgets.

JQO

In addition to functions in the standard JQuery libraries, there are thousands of additional 'plugin' JQuery scripts. This namespace is a repository for those plugins which we found useful enough to include in the standard distribution of MiServer.

Base64

Base64 is an encoding scheme used to represent binary data as an ASCII string for transmission via HTTP request. It was developed as a way of encoding binary data sent through email although its usage is more general in today's environment. This namespace contains functions which both encode and decode text as Base64.

JSON

JavaScript Notation Format, or JSON, is a data-interchange format which is used to transmit data to and from JavaScript. This namespace contains functions which converts JSON, as a character vector, to and from any of 3 forms:

- APL
- XML
- Namespace

Both the APL and XML forms are 100% lossless, which means that:

- json ≡ APLtoJSON JSONtoAPL json and
- json ≡ XMLtoJSON JSONtoXML json

The namespace representation may lose data when an object name contains characters that are not valid in APL names.

CHAPTER 6

Getting Stylish

6.1 Styles

Up until now, we have focused on a MiSite's functionality, but presentation can be an important factor in a MiPage's usefulness. HTML says little about how the page will be visually represented. Browsers apply a number of properties, also called styles, to the HTML which inform the way the browser will display the content.

Styles control the look and feel of every HTML element with a visual representation. There are dozens of types of styles, including ones that affect:

- The font
- A page's background color
- The thickness of lines in a table
- The way a text is processed by accessibility programs that read web pages to visually impaired users.

This section will explore a number of ways to control the styles in your MiPage, and will look at creating a consistent look and feel for your MiSite.

Style Basics

There are three main ways to exercise additional control over the look and feel of your website:

- Using the style attribute within a tag
- Inserting a "cascading style sheet" (CSS) within the `<head>` tags of an HTML page
- Linking an external CSS document to an HTML page

A style has two parts:

- Selectors – one or more reference to possible HTML elements
- Declarations - one or more property-value pairs.

If two styles affect the same property on the same element, the last style furthest down the page is the value used when the page renders.

Selectors

When not inserting styles directly into a tag, you must identify the HTML elements which will be affected by a set of declarations. Selectors are references to possible HTML element names or attributes. If elements that match a selector's specifications exist, then the declarations associated with that selector will be applied to those HTML elements. If they do not exist, the declarations are simply not applied to anything

Each different type of selector requires a particular syntax. A few examples of selector syntaxes include:

Syntax	Description	Example
`TagName`	A tag selector This selector affects each instance of a particular tag.	`h1` – selects all `<h1>` tags `<h1>Title</h1>`

`.ClassName`	A class selector
	Preceded by a period, this selector affects each instance of the named class.
		`.tree` – selects all HTML elements have the attribute "class" containing the value "tree." Classes are used to describe multiple elements.
		
		`<span class="tree">`
		`Tree</span>`

`#IdName`	An id selector
	Preceded by a hash, this selector selects the page element with the named id attribute.
		`#table` – selects the page element with the attribute "id" containing the value "table". The id attribute is intended to refer to a specific element and it is recommended that each element id being unique.
		
		`<div id="table">`
		`Table</div>`

A few notes on selectors:

- Each of these selectors can be combined with other selectors to pinpoint more specific elements. For example: `h1.note {color:red;}` – applies to all `<h1 class="note">` tags
- Multiple selectors can be used at once, applying a set of declarations to each selector. For example: `h1 h2 h3 h4 {textdecoration:none;}` – applies its declaration to the `<h1>`, `<h2>`, `<h3>` and `<h4>` tags

Declarations

Declarations are the ‘what to do’ of the style. Each declaration sets the value of one or more ‘properties.’ There are dozens of these properties, many of which have a unique set of valid answers. Declarations consist of one or more property value pairs, where a colon separates the properties and values of each pair and each pair ends in a semi-colon. Below are examples of a few properties and valid declarations. A more complete list can be found at `http://www.w3schools.com/cssref/default.asp`.

Property	Description	Example Declaration
`color`	Specifies the color of the text	`color: blue;`
`text-align`	Specifies the justification of the text (left, right or center)	`text-align: center;`
`float`	Specifies if an element is ‘floating’ and what direction that will be in	`float: left;`
`margin`	Specifies the distance between the border of the HTML element and other elements	`margin: 0 auto;`
`padding`	Specifies the distance between the border of the HTML element and content within that element	`padding: 10px;`

6.2 In-Line Styles

While not generally recommended, you can insert styles directly into a tag, via the `style` attribute. This bypasses the need for a selector, as it will affect that tag, and its contents, alone.

```
    'style' 'color:red;' #.HTML.div 'This text will be red'
<div style="color:red;">This text will be red</div>

    'style' 'background-color:blue;' #.HTML.div 'My background is blue'
<div style="background-color:blue;">My background is blue</div>
```

6.3 Cascading Style Sheets

Cascading Style Sheets (CSS) are collections of styles, called rules, which affect an entire page. Each rule has two parts:

```
selector { property: value; property: value ; }

or

selector {
 property: value;
 property: value ;
}
```

Associating CSS with a web page

There are two ways to associate a style sheet with your web page.

- Internal Style Sheets – CSS is within the head tags
- External Style Sheets – CSS is on a separate file that your page

Internal CSS

CSS can be placed in a page's `<head>` tags by wrapping the entire style sheet within `<style type="text/css"></style>` tags.

Styles Formatted for the <head> Tag

```
<style type="text/css">
      body { color: blue; }
      #Content { opacity:0.7; }
      .center { text-align: center; }
</style>
```

Example: An HTML Page with Internal CSS

```
<html>
      <head>
            <style type="text/css">
                  body { background-color:lightgray; }
                  .big { font-size:200%; }
            </style>
      </head>
      <body>
            Styling documents is <span class="big">easy</span> and <span
class="big">fun</span>.
      </body>
</html>
```

External CSS

The industry standard is to keep the CSS in external documents linked to your page. External CSS files allow you to tweak CSS between page loads and even swap whole CSS files for others, which is useful for accessibility concerns. Moreover, you can have a standard set of styles that affects an entire site, creating a consistent look and feel.

Simply gather your styles onto a text file and save it with a .css extension. Then, associate that style sheet with your webpage by placing a `<link rel="stylesheet" type="text/css" />` tag within the `<head>` tags. The `<link>` tag must point to the location of your style sheet using the `href` attribute. The below tag is referencing the style sheet `SiteRoot/Styles/Style.css`.

```
<link href="/Styles/style.css" rel="stylesheet" type="text/css">
```

Let's take another look at the internal CSS example formatted as external CSS. The below example will produce the same browser representation as above.

Example:

SiteRoot/Styles/style.css

```
body {
background-color:lightgray ;
}

.big {
font-size:200% ;
}
```

An HTML page with External CSS

```
<html>
      <head>
            <link href="/Styles/style.css" rel="stylesheet" type="text/css">
      </head>
      <body>
            Styling documents is <span class="big">easy</span> and <span
class="big">fun</span>.
      </body>
</html>
```

6.4 Ideas for Using Style Sheets in MiPages

By default, DemoServer.Wrap (MildServer.Wrap was discussed in section 4.3) associates each MiPage with `SiteRoot/Styles/style.css`. This file contains some basic formatting and browser compatibility styles, as well as the styles that produce the theme you see among the demo site's pages.

If you have a different style sheet you would like to associate with the site, you can do one of three things.

- Replace style.css with another file of the same name
- Change the base style sheet path in DemoServer.Wrap
- Pass the file path of a new style sheet to the Style method of the request object (req.Style). This appends the link for a new style sheet with that path after the `style.css` link

CHAPTER 7

Under the Covers

7.1 Going Deeper

By this point, you should already know how to construct basic MiSites and MiPages. You can generate basic text pages or pages which communicate with the server. However, there is a lot going on behind the scenes, specifically regarding server-browser communication. Also, this chapter will look at another MiServer resource, HTTPRequest, which is responsible for gathering the request sent by the browser, parsing it into useful chunks and generating the response.

7.2 Browser to Server Communication

GET and POST HTTP Requests

In general, browsers communicate with web servers by sending HTTP requests for resources. Data must be sent the same way, along with a request. To allow for different types of communication, there are a few different HTTP request structures. Currently, MiServer only supports GET and POST requests.

- GET requests are the most common form of HTTP request. These are generated when an URL is entered into an address bar or a link is clicked. It is possible to encode data along with the URL as name-value pairs. As the data is visible in the URL bar, it is generally considered less secure.
- POST requests are generally used by “forms”, and contain name value-pairs in the body of the request. Since the data transfer happens in the background, this method is generally considered more secure.

An example that captures the interactions of HTTP requests of different types is included with the Demo Server, in `Demo/Admin/httprequest.dyalog`. If you have default server settings, this folder will require user authentication to be accessed by your browser. If a username and password are requested and you have not yet set up the SimpleAuth extension, the default administrator username and password are ‘admin’ and ‘admin.’.

Getting Data to the Server

The GET Request and Data Encoded URLs

Whenever you put a URL in an address bar and access a page, you are sending a GET HTTP request. Along with the resource path, data can be passed with this request as well, in the form of name value pairs. These pairs are appended to the end of the resource path after a question mark. Names and values are separated by equal signs and pairs are separated by ampersands.

Examples:

```
http://localhost:8080/reverse?Name=Beethoven

http://localhost:8080/reverse?Name=Beethoven&Action=Submit
```

These values are sent to the page for the server and associated scripts to use.

Since there are characters that have special significance in the address field (“/” or “:” for example), values that include those special characters that would affect the parsing of the URL. To have a faithful representation of the data, URL can be encoded to represent these symbols by a percent symbol followed by a two digit hexadecimal number representing the Unicode code point of the symbol. Thus “/” is represented as %2F and the “:” as %3A. This is true for most characters, with few exceptions. The very common space character is one of those exceptions, in that in addition to being represented by %20, it can also be represented by an unencoded “+” symbol.

An HTMLInput.Enclose function generating that page link:

```
    text ← 'Reverse Page with Beethoven in the Edit Field'
    'a href="http://localhost:8080/reverse?Name=Beethoven"' HTMLInput.Enclose text

<a href="http://localhost:8080/reverse?Name=Beethoven">Reverse Page with Beethoven in the Edit Field</a>
```

When the request is sent, data in the URL line can be found in the Arguments HTTP header element. req.Arguments contains an N x 2 matrix of the pairs, where N is the number of pairs.

Forms and the Post Method

An HTTP POST request is another way to send data back to the server. It can be triggered by a form element, a section of an HTML document surrounded by <form> tags. A form is designed to contain input elements and a control which initiates the submission of a request containing the data of those input elements.

The form tags specify the resource requested upon submission (action="/reverse.dyalog"), the type of HTTP request (method="post") and the content encoding type, which needs to be specified but you fortunately do not need to understand (enctype="multipart/form-data").

Form from `Reverse.dyalog` page source

```
<form action="/reverse.dyalog" method="post" enctype="multipart/form-data">
    <br />Enter Text:
```

```
    <input type=text size=10 id="Name" name="Name" value="Beethoven">
    <br /><br />
    <input type="submit" name="Action" value="Reverse" >
    <input type="submit" name="Action" value="Clear" >
</form>
```

Input objects

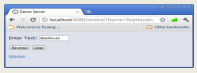
HTML has a number of tags that are recognized as data when within a form during submission. Each of these tags has the attributes `name` and `value`, which populate the data portion of the submission. Some of these elements are used to allow for users to submit data to the server. These include, but are not limited to, the `<textarea>` tag, the `<select>` tag and variations of the `<input>` tag.

While describing the complete functionality of these tags is outside the scope of this manual, we will make use of the `text` and `submit` types of the `<input>` tag.

In the case of the `<input>` tag, the `type` attribute allows developers to select from a number of different types of input controls, like text boxes, check boxes or buttons. When `type="text"` the `<input>` tag renders as an editable text box. When `type="submit"`, the tag renders as a button.

A text box with the text “Beethoven”

```
    'Name' HTMLInput.Edit 'Beethoven'
<input type=text size=10 id="Name" name="Name" value="Beethoven" >
```



A submit button with the text “Reverse”

```
    'Action' HTMLInput.Submit 'Reverse'
<input type="submit" name="Action" value="Reverse">
```

Submitting a Form

When clicked, a submit button within a form will initiate a HTTP request described by the form tags. The browser then gathers the data of all elements that are ‘valid’ for submission. A valid submission has a control name, which is usually its `name` attribute, is paired with the current value of the element. These are gathered by the browser and sent via the HTTP request.

7.3 How MiServer Gets Client Data: HTTPRequest

When MiServer receives a resource request, it creates a new instance of the class HTTPRequest. This class parses the information contained in the request into a number of fields. The request object is then passed to the MiPage and used to format the response.

There are three places that data encoded in the request may be found, depending on how it was passed to MiServer: (Remember, the request object is always passed to the Render method of a MiPage as the right argument req.)

- req.Data - an N x 2 array of the pairs sent within the body of a POST request
- reg.Arguments - an N x 2array of the pairs encoded in the URL
- Similarly named public fields - If a MiPage contains a public field with the same name as the first element of a name-value pair in req.Data or req.Arguments, that field will be set to the value of that pair. For example, `Reverse.dyalog` produces an `<input>` tag with `Text` is its name attribute. When the server receives the value of that input tag, it sets the public field Text to the element's value before calling the Render method

HTTPRequest also constructs the HTTP response. A character vector of HTML is passed to req.Return at the end of the Render method which stores it in req.Response.HTML. A reference for the HTTPRequest class is available in Appendix G.

CHAPTER 8

MiServer Skins

8.1 Customizing your MiServer

An installation of MiServer can host multiple sites, each with completely different behaviors. To accomplish this, MiServer is fully editable and designed with extension in mind. Almost every component of the MiServer is stored as a `.dyalog` scripted file, meaning that core files and user generated files alike can be modified both in the Dyalog session and from most text editors.

While we encourage everyone to explore the core files, it is likely that you will be using the core capacities for server function with some site specific behavior modifications. As we have seen with MiPages, the object oriented nature of MiServer will come in handy for this with the 'MiServer skin' a class derived from the MildServer class. We will look at the development and implementation of a MiServer skin.

8.2 MiServer Skins: Spicing up the MildServer class

The MildServer base class contains the core server functionality. You can set the `ClassName` parameter within `Server.xml` to `MildServer` to use its server behaviors but as the name suggests, it is a 'mild' version of the server has only very basic functionality.

More complex, and frankly more useful, servers require a bit of forethought and the use of a class derived from the MildServer base class. This is called the MiServer 'skin'. The skin can override one or more methods in MildServer, which modifies the behavior of your server without touching the core code.

The following sections are descriptions of the overridable methods, including the behaviors in the MiServer skin, DemoServer.

Session Handling

If session handling is enabled, as described in chapter 8, onSessionStart should perform any processing necessary when a new session is created. Similarly, onSessionEnd should perform any processing when a session ends. The session handler packaged with MiServer, SimpleSessions, calls these functions.

DemoServer.onSessionStart and DemoServer.onSessionEnd simply produce log entries.

HTML Wrapping

MildServer.Wrap takes the HTTP request object after it has passed through the MildPage and performs final processing on the response. While it is possible to set up a MiSite so that each page generates the entire HTML document, Wrap can be used to implement a consistent look and feel across all pages of your MiSite.

DemoServer.Wrap takes the HTML which was passed to the request object at the end of the MiPage's Render method and wraps it with the body tags of an HTML template, as well as creates a common, but overridable <head> tag structure that associates all its MiPages with the default style sheet.

Error Handling

MildServer.Error allows for custom, server error trapping behaviors.

DemoServer.Error logs errors and posts a server-side error message to the browser.

Logging

MildServer.Log is a function for capturing and organizing log messages.

Several methods in the MiServer architecture pass their status messages to Log, along with a number indicating the type of message. MildServer.Log posts these messages to the session, and has a configuration parameter in `Server.xml, LogMessageLevel`, which determines which logs are displayed.

This parameter is either set to 0 for no messages, ¯1 for all messages, or the sum of some or all of the following five message levels, indicating which are displayed:

- 1 - error/important
- 2 - warning
- 4 - informational
- 8 - transaction (GET/POST)
- 16 – compression related

Cleanup

MildServer.Cleanup is called in the MildServer class destructor. Depending on the functionality of your website, there may be operations you need to perform as the server shuts down, such as untying files, disengaging from a database or even shutting down programs that were started to assist the server.

Idle Behavior

MildServer.OnIdle is designed to allow the server to respond to sessions which have been idle. After a period of minutes, set by the `Server.xml` parameter `IdleTime`, OnIdle is called. This is useful for conserving server resources.

A MiServer Skin: DemoServer.dyalog

Below is the example server included in SeverRoot/Demo/Code. Notice that it does not override MildServer.Error, deferring to the behavior in the base class.

```
:Class DemoServer : MildServer

    NL←⎕UCS 13 10

    :Include #.HTMLInput

    ∇ make args;file
      :Access Public
      :Implements Constructor :Base args
      :If #.Files.Exists file←Root,'/config/DataSources.xml'
          Datasources←(#.XML.ToNS #.Files.GetText file).Datasources
          :If 0=#.⎕NC'SQA'
              'SQA'#.⎕CY'SQAPL' ⍝ retrieve SQA if needed
          :EndIf
      :EndIf
    ∇

    ∇ onSessionEnd session;msg
      :Access Public Override
      :With session
          msg←'End of session ',(⍕ID),': User=',User,'; Last active: ',,#.Dates.TSFmtNice LastActive
      :EndWith
      4 Log msg
    ∇

    ∇ onSessionStart req
      :Access Public Override
      4 Log'New session ',(⍕req.Session.ID),' started to process ',req.Page
    ∇

    ∇ level Log msg;report
      :Access Public Override
      report←Config.LogMessageLevel ⍝ report levels specified in config (use 0 for no reporting, ¯1 for
                all)
      :If report bit level
          ⎕←msg
      :EndIf
    ∇

      bit←{⎕IO←0  ⍝ used by Log
          0=⍺:0 ⍝ all bits turned off
          ¯1=⍺:1 ⍝ all bits turned on
          (⌈2⍟⍵)⊃⍵⍴(1+⍵)⍴2)⊤⍺}

:EndClass
```

CHAPTER 9

Extensions

9.1 Extensions

MiServer Extensions are classes or namespaces which are used to extend the functionality of MiServer. Extensions often require infrastructure within MiServer to function. These hooks into the server are generalized, however, to allow new extensions to be easily developed and swapped in. Extensions are stored as scripted files in `ServerRoot/Extensions` and do not have a specific format.

MiServer ships with the following extensions:

- SimpleSessions.dyalog – implements and handles sessions
- SimpleAuth.dyalog – implements authentication for user access
- ContentEncoder.dyalog – provides a framework for content encoding extensions
  - deflate.dyalog – implements the 'deflate' encoding, a common encoding/compression scheme
- Lumberjack.dyalog – logs HTTP requests

9.2 SimpleSessions - A Session Handling Extension

The HTTP protocol specifies that HTTP requests and responses are 'stateless.' Servers treat each request as if it came from a unique client. Information must be imbedded in requests and responses that allow for web servers to recognize patterns in requests and identify users. These patterns, called sessions, provide a context for data to persist over multiple page loads. Sessions are ideal for sites that require users to identify themselves with a name and password or that have information that must be available during an entire visit, like a shopping cart.

SimpleSessions is a basic extension to MiServer that identifies and handles sessions.

Cookies

SimpleSessions uses a mechanism called a 'cookie' to identify a unique session. A cookie is an item of data that a browser is instructed to include in the 'Cookies' header of the HTTP request. A cookie has the following attributes:

- The name of the cookie
- The value of the cookie
- Which site and site paths the cookie will be sent with the HTTP request
- The amount of time that the cookie will “persist,” after which it will be deleted

SimpleSessions checks to see if the HTTP request contains a cookie named ‘Session.’ If does not, it commands the browser to create one and with a randomly generated “id” that is associated with a new session. This session will be recognized until it times out (by default, after 10 minutes).

If the extension finds a ‘Session’ cookie, it compares the cookie’s value to the list of values associated with sessions which have not yet timed out. If it finds a match, the session’s timeout clock is reset and the session data is copied to the req.Session namespace.

Using Cookies in Your MiPages

Cookies can persist for months, maintaining a user’s data even after weeks of inactivity on a site. This means data can last through server and client disconnections or crashes.

The request object contains methods that edit the HTTP response to include instructions for setting and deleting cookies, as well as a function for retrieving cookie values:

- req.SetCookie – Adds a `set-cookie` header to the HTTP response. This includes the cookie’s name, value, the request paths on which the cookie will be sent and the cookie’s expiration date
- req.DelCookie – tells the browser to delete a cookie by name
- req.GetCookie – returns the value of a cookie

Using Cookies and In-Session Data Storage: An Example

The following MiPage demonstrates storing and retrieving information from cookies and session variables.

```
:Class CookieSessionVarExample: MildPage
   :Include #.HTMLInput

   :Field Public Action← ''
   :Field Public Text←''
   :Field Private Display←''


 Render req;html
    The MiPage contains:
    A header which will display content when appropriate
    An edit field for entering text
    Several self explanitory submit buttons

     :Access Public

     DoAction
```

```
    html←2 BRA'h1'Enclose'Display: ',Display
    html,←BRA'Text'Edit''
    html,←BRA'Action'Submit'Display Field Text'
    html,←BRA'Action'Submit'Save Text to Session Variable '
    html,←BRA'Action'Submit'Display Session Variable Data'
    html,←BRA'Action'Submit'Save Text to Cookie'
    html,←BRA'Action'Submit'Display Cookie Data'
    html,←BRA'Action'Submit'Reset Data'
    html←('post'Form)html

    req.Return html



DoAction
    ⍝ If the Session variable does not exist, create one.
    :If 0=req.Session.State.⎕NC'DataToDisplay'
        req.Session.State.DataToDisplay←''
    :EndIf


    :Select Action
    :Case 'Display Field Text'
        Display←Text
    :Case 'Save Text to Session Variable'
        req.Session.State.DataToDisplay←Text
    :Case 'Display Session Variable Data'
        Display←req.Session.State.DataToDisplay
    :Case 'Save Text to Cookie'
        req.SetCookie'TestCookie'(Text)'/svc' 1
    :Case 'Display Cookie Data'
        Display←req.GetCookie'TestCookie'
    :Case 'Reset Data'
        Text←''
        req.DelCookie'TestCookie'
        req.Session.State.DataToDisplay←Text
    :EndSelect



:EndClass
```

9.3 SimpleAuth – An Access Control Extension

SimpleAuth.dyalog is a simple access control extension which can restrict access to site resources based on user credentials. The configuration file `Access.xml` can be found in the `SiteRoot\Config\` directory. Each `Folder` element contains a path element and a group element. The path determines the path of the directory being restricted. The `Groups` element is a

comma separated list of all the user groups allowed access to that directory. “**” is the wildcard for either element.

A Sample Access.xml

```
<Access>
   <Folder>
      <Path>/Admin</Path>
      <Groups>admin</Groups>
   </Folder>
   <Folder>
      <Path>/LocationOfTheFellowship</Path>
      <Groups>admin, fellowship</Groups>
   </Folder>
  <Folder>
       <Path>/Mordor</Path>
      <Groups>admin, management, ringbearer</Groups>
   <Folder>
      <Path>**</Path>
      <Groups>**</Groups>
   </Folder>
</Access>
```

When a browser requests a resource from a restricted directory, the client is prompted to enter a username and password. If the credentials match a user defined in the configuration file, `SiteRoot\Config\Users.xml`, those user’s credentials are added to the session. Each user is associated with one or more groups, noted in the `Groups` element of their `Users.xml` entry. If the user is associated with one of the groups with access to the directory, the page loads as normal. Otherwise, an access error page is displayed.

A Sample Users.xml

```
<Users>
   <User>
      <ID>gandalf</ID>
      <Pass>youshallnotguessmypassword</Pass>
      <Groups>admin, fellowship</Groups>
   </User>
   <User>
      <ID>fbaggins</ID>
      <Pass>goonwithoutmesam</Pass>
      <Groups>ringbearer, fellowship</Groups>
   </User>
      <ID>gimli</ID>
      <Pass>shortiscute</Pass>
      <Group>fellowship</Group>
   </User>
   <User>
```

```
      <ID>Gollum</ID>
      <Pass>precious</Pass>
      <Group>ringbearer</Group>
   </User>
   <User>
      <ID>sauron</ID>
      <Pass>allseeingeye</Pass>
      <Group>management</Group>
   </User>
</Users>
```

Please be aware that SimpleAuth stores passwords in plain text in an XML file.

9.4 Lumberjack – An HTTP Request Logging Extension

Lumberjack is a logging extension packaged with MiServer, which provides basic logging of resource requests. A log record is made for every HTTP request. The data logs can be used to analyze web site usage, profile resources usage, and detect suspicious patterns of activity.

A sample Lumberjack log

```
127.0.0.1:49330 - [08/Jun/2012:22:23:58 +0000] "get  /index.dyalog"
200 164 3879 1638
```

Each log consists of the following. If an element does not have a value, it is returned as a dash:

1. <IP address>:<Port> – ex. 127.0.0.1:49330
2. <User ID>
3. [<Timestamp in UTC>] – ex. [08/Jun/2012:22:23:58 +0000]
4. "<HTTP command> <Resource>" – ex. "get /index.dyalog"
5. <HTTP status code> – ex. 200 – see the w3
6. <Time to send response> – in milliseconds – ex. 164
7. <Size of response before compression> – in bytes – ex. 3879
8. <Size of response after compression> – in bytes – ex. 1638

Lumberjack works in conjunction with the configuration file, Lumberjack.xml, which must be in your site's Config directory. Four parameters can be set in the file:

- `active`
- `directory`
- `interval`
- `prefix`

If the `active` parameter is set to 1, Lumberjack starts with the server. It works by recording a log for each HTTP request. These requests are then written to a log file, one of which is created each day,

in a directory specified by the `directory` parameter. These paths can be made relative by prepending them with `%ServerRoot%` or `%SiteRoot%`.

To save server resources, these logs are cached and then written to the log file at an interval specified in seconds by the `interval` parameter. Note that in the event of a server crash, any cached logs will be lost.

When Lumberjack writes a log, the file is named with the following format: `YYYYMMDD.txt`. If the `prefix` parameter has a value, it is prepended to the filename. For example, if the value of the `prefix` parameter was "`misite`" a log made on 8 July, 2012 would be named `misite20120708.txt`. If you have several MiServers writing Lumberjack logs to the same directory, you would be well advised to specify a unique prefix for each server.

A sample Lumberjack.xml

```
<Lumberjack>
  <active>1</active>  <!-- 1 for yes, 0 for no -->
  <directory>%ServerRoot%/ServerData</directory>
  <interval>10</interval> <!-- in seconds -->
  <prefix>misite</prefix>
</Lumberjack>
```

CHAPTER 10

Using Relational Databases in a MiSite

10.1 Using External Data Sources

Many webpages display data which comes from an outside source. Displaying today's weather forecast or showing you the value your retirement portfolio requires that communication between the page and a database of some kind. The page will obtain data, possibly manipulate it, format it using HTML and present it to the client (and perhaps allow the user to update the information).

This data can come from a variety of sources including:

- Text Files
- Dyalog Component Files
- Relational Databases
- CSV Files
- Excel Spreadsheets

10.2 Relational Databases

A data driven website needs someplace to store its data. Often, this is in a relational database, such as MySQL, Microsoft Access, IBM DB2, Microsoft SQL Server or Oracle.

Interacting with Relational Databases

Open DataBase Connectivity (ODBC) is a cross platform, language independent interface and is the most widely used standard through which programs interact with databases. SQAPL is Dyalog's ODBC interface which provides access from Dyalog APL to any ODBC compliant database. It is a standard component of Dyalog APL under Windows and is available as an option on other platforms.

MiServer contains SQL, a namespace of utility functions that simplifies SQAPL integration.

Setting up Datasources.xml

SQL requires an XML configuration file called Datasource.xml, contained in the SiteRoot/Config/ directory. The file contains 0 or more `Datasource` elements which are information used by SQL to identify and connect to ODBC data sources.

Each Datasource element can be defined by five possible elements:

- Name – The name used within MiServer to refer the datasource
- DriverOptions – SQL driver options
- DSN – Database Source Name
- User – User name for authentication in the database
- Pwd - Password for authentication in the database, although we do not recommend keeping your password in a text file on your computer

In order to use SQL, you need to define one or more datasources. The Name element is required, along with a way to locate the database. The location information can kept either as a Database Source Name (DSN) as defined in your computer's datasource administrator in the DSN element, or you can specify how connect to the database if it is a DSN-less connection in the `DriverOptions` element.

If these concepts are unfamiliar to you, please read the SQAPL manual, which you can find at `http://docs.dyalog.com`. The `datasources.xml` file included with the Demo server can be found below, including two properly defined datasources:

Datasources.xml

```
<Datasources>
 <Datasource>
  <Name>ZipCodes</Name>
  <DriverOptions>DRIVER={Microsoft Access Driver (*.mdb, *.accdb)};
          DBQ=c:\MiServer\dyalog2011\data\zipcodes.accdb;ExtendedAnsiSQL=1;MaxBufferSize=2
          048;</DriverOptions>
 </Datasource>
 <Datasource>
  <Name>SQRTest</Name>
  <DSN>SQRTest</DSN>
 </Datasource>
</Datasources>
```

When Boot initializes MiServer, it looks for data sources defined in Datasources.xml. If there are any sources defined, and the SQA namespace does not already exist, it copies in and initializes SQAPL.

SQL.ConnectTo and SQL.Do

Once a datasource reference has been established, the SQL namespace makes use of them to bring data to your page.

SQL.Do is a cover function for SQAPL.Do, which executes SQAPL queries. SQL.Do connects to the referenced datasource, performs a query and then closes the connection. When Do queries for data, it returns a namespace, which contains the variables Columns, Data and ReturnCode.

- Columns are the column names of the query's result
- Data is an N × M matrix of data returned as a result of the query
- ReturnCode will be 0 if the query was successful, all other results being error numbers described in the SQAPL manual

You can also use other functions from SQAPL directly and still use `Datasources.xml` to define your sources. SQL.ConnectTo initiates a database connection, taking the Name element of a Datasource element as its right argument. After the connection has been established, you are free to use SQAPL as you please. The SQL namespace function reference is in Appendix C.

Example:

```
:Class SQLdemo : MiPage

   :Include #.HTMLInput
   :Include #.SQL

   :Field Public state←''
   :Field States


 Render req;HTML;form;data;chunk
    :Access Public
     html←'h2'Enclose'SQAPL/JQuery Demonstration'
     :If 0∈⍴States
        html,←BRA'h3'Enclose'ZipCodes database is not available!  Sorry...'
     :Else
        chunk←BRA'h3'Enclose'Zip Codes by State'
        form←'Select State: ','state'DropDown States state'autofocus="autofocus"
              onChange="this.form.submit()"'
        chunk,←'action="#"'('post'Form)form
        :If state≢''
           data←Do'ZipCodes' 'select * from ZipCodes where StateAbbr = :a<C2: order by Zipcode'
              state
           :If 0=data.ReturnCode
              chunk,←req #.JQ.TableSorter'tab1'(data.Columns⍪ data.Data)'' 1
           :Else
              chunk←'h3'Enclose'Database query failed? RC = ',⍕data.ReturnCode
           :EndIf
        :EndIf
        html,←chunk
     :EndIf
     req.Return html




 Init;data
     :Implements constructor :base
     :Access public
```

```
    States←''
    data←Do'ZipCodes' 'select * from States order by StateName'
    :If data.ReturnCode=0
       States←'' ''⍎data.Data
    :EndIf



:EndClass
```

10.3 Displaying Data in your MiPage

Using data received from the browser is simple. Remember that you can collect data from:

- Public fields the named the same as the `id` attribute of input elements that are 'valid' for submission
- req.Data - if the HTTP post method was used to send data in the body of the HTTP request
- req.Arguments - if data was encoded in the URL

Since data passed through the HTTP request is received as character vectors, populated fields can often simply be added to a vector of HTML.

```
html ← 'Here is the data from the Text field: ', Text
```

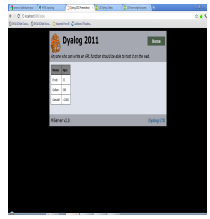
Displaying Tables

If the shape of your data is little more complex, you may think about organizing it as a grid. HTMLInput.Table is used to display a simple matrix of data by enclosing the data within a `<table>` tag structure. The first elemaent of the right argument passed to HTMLInput.Table is a matrix of no greater than depth 2. Please see this function's reference entry in Appendix A for more information.

```
:Class table : MildPage

   :Include #.HTMLInput


 Render req;tabledata;html
    :Access Public
    tabledata←4 2⍴'Names' 'Ages' 'Frodo' 33 'Gollum' 589 'Gandalf' '~2000'
    html←Table tabledata '' '' '' 1
     req.Return html



:EndClass
```

Here's what this might look like with the below styling applied:

```
    CellAtts ← 'style="padding:10px;border: 1px solid black;background-color:white;"'
    HeaderAtts ← 'style="padding:10px;border: 1px solid black;background-color:gray;font-
weight:bold;"'
html←Table tabledata '' CellAtts HeaderAtts 1
```



If we were a bit smarter about that, we would add an `id` attribute (the optional left argument of the function) and give the headers and cells classes to be referenced by a CSS document.

```
    html←'Ages' Table tabledata '' 'class="cell"' 'class="header"' 1
```

Then we append the below text to a CSS document that is associated with the page.

```
#Ages.cell{
padding:10px;
border: 1px solid black;
background-color:white ;
}

#Ages.header{
padding:10px;
border: 1px solid black;
background-color:gray ;
}
```

And we get the same result, only with more manageable styling and cleaner code.

Advanced Tables

If you are dealing with large chunks of data, you may benefit from a table that incorporates sorting and pagination. You may be interested to look at JQUI.TableSorter which is described with examples in the next chapter.

CHAPTER 11

Improving your UI with jQuery

11.1 jQuery and jQueryUI

When web designers do something that makes their website ‘sexy’ they are usually talking about one of a handful of tools, JavaScript being one of the most popular.

JavaScript is a scripting language that is compatible with most browsers. As a client side scripting language, it works by having its code executed by the browser instead of by the server. This allows pages to change without having to send a submission.
It is used in many corners of the web, creating dynamic user experiences, adding visual effects and animations and even supporting server side business logic by processing or validating data before it is sent back in a request.

However, JavaScript is a complete and at times obtuse languag
average web developer, a number of JavaScript libraries came
top as the most widely used. jQuery is a comprehensive, open s
extensive control over the document object model (DOM). jQu
client side scripting because it is free to use and develop, has a
to learn.

The Document Object Model, or DOM, is the tree-like representation of the HTML elements in a web page, in the form of nested objects. JavaScript can manipulate this structure, modifying the position of an object within the DOM.

Most of the following functions use jQueryUI, a library that co
jQuery library which affect the user interface. It can be used to create effects and animations but also comes with a number of prebuilt, easily insertable code snippets called widgets. jQueryUI makes it easy to develop sophisticated user interactions. It is a powerful library and frankly it’s really cool.

Also, one of the benefits of such a large user base is the number of community developed plugins in addition to the official jQuery releases. Many of these are well documented and have the backing of an active user community.

For more information about jQuery and jQueryUI, visit `http://www.jquery.com` and `http://www.jqueryui.com`.

11.2 The MiServer–jQuery Interface

We developed a few functions to integrate jQuery into your MiPages, designed to allow for basic implementation with a minimal understanding of JavaScript. However, they are not comprehensives interfaces, and the optional parameters that can be passed to jQuery will need to structured as described in the next section.

There are three namespaces which contain these widgets:

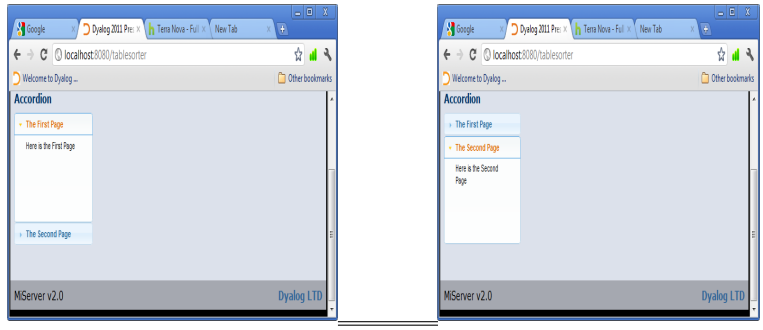
- JQ – interfaces to the jQuery library
- JQUI – interfaces to jQueryUI widgets
- JQO – interfaces to third party jQuery plugins

The following is a description of the more useful widgets and functions of JQUI and JQO. For a complete reference of these namespaces, see Appendix B.

JQUI.Accordion

This widget takes multiple sections of HTML and condenses them into a single object. It has a number of tabs which are clicked through to change the content selections. In the default configuration, the tabs appear to slide to make way for the new content.

```
content←2⍴⊂''
content[1]←⊂'Here is the First Page'
content[2]←⊂'Here is the Second Page'
headers←'The First Page' 'The Second Page'
jqueryPars←'fillSpace: true'
accordion← req #.JQ.Accordion'myaccordion' headers content jqueryPars
html←'style' 'width:200px; height:200px;'#.HTML.div accordion
```



JQUI.DatePicker

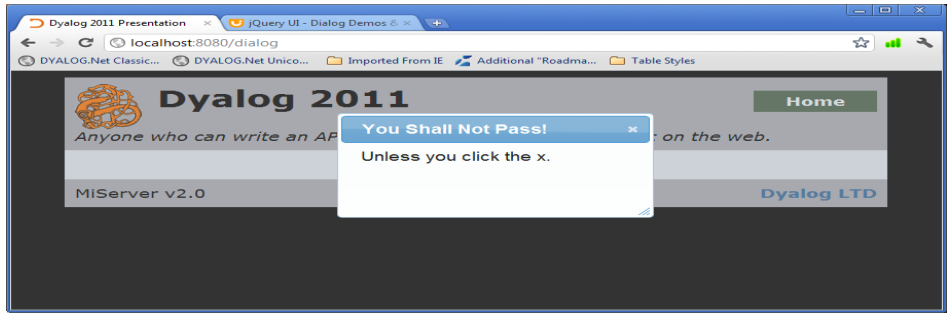
This widget is a text input box that pops up a calendar when selected. When a date is selected from the calendar, the value is placed in the text box.

```
dateID←'mydate1'
editPars←'' 30
jqpars←'changeMonth: true,changeYear: true,dateFormat: "DD, d MM yy"'
html←req #.JQ.DatePicker dateID editPars jqpars
```

JQUI.Dialog

This widget creates a ‘pop up window’ from the contents of a `<div>` tag element. It does not open a new browser window with the contents, but the default configuration creates a draggable box with a title and an exit button in the upper right hand corner. This box can also be modal as seen in the example below.

```
title←'You Shall Not Pass!'
contents←'Unless you click the x or another window.'
jqpars←'modal:true'
html←req #.JQ.Dialog 'dialog' title contents jqpars
```

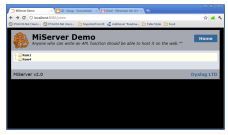


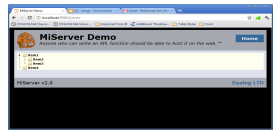
JQO.jsTree

This widget is takes a number of items and, based on the nesting levels described in the second parameter, represents the data as a collapsible tree.

Please note that an item cannot be more than one level deeper than the preceding item. Also, while the initial level is arbitrary, no item may ever be less deep than the first item.

```
levels←1 2 2 1
items←'item'∘,¨⍳¨↑⍴levels
html←req #.JQO.jsTree 'tree' levels items
```



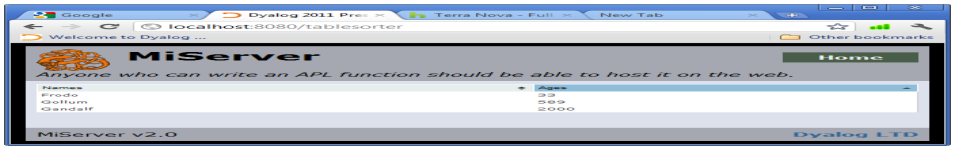


JQO.TableSorter

This widget is a table that is sortable by column with optional pagination. It is possible to sort multiple columns hierarchically, by selecting one to sort then selecting the next. The second column will be sorted in the context of the first.

If the pagination plugin is active, then any items over ten will be hidden on a different ‘page.’ The user can change pages and select the number of items per page.

```
data←4 2⍴'Names' 'Ages' 'Frodo' 33 'Gollum' 589 'Gandalf' '2000'
tableID←'mytable'
tablePars←(data'' '' '' 1)
jqpars←''
html←req #.JQ.TableSorter tableID tablePars jqpars
```



JQUI.Tabs

This widget presents multiple sections of content as a single object that can be navigated by a list of ‘tabbed’ heading across the top of the object. You may notice that this bears some resemblance to the properties dialogs used by Windows programs.

If instead of HTML a path is passed as one of the tab contents, Tabs will load the contents of that resource into the active tab.

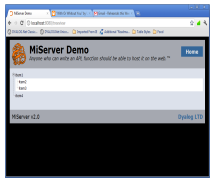
```
id←'tabs'
tabnames←'tab1' 'tabs'
content←'contents of tab 1' 'HTML/tab2.html'
jqpars←''
html←req #.JQ.Tabs id tabnames content jqpars
req.Return 'div style="width:150px;"' #.HTMLInput.Enclose html
```

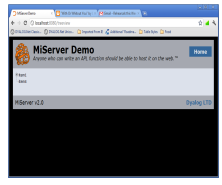
JQO.Treeview

This widget is takes a number of items and, based on the nesting levels described in the second parameter, represents the data as a collapsible tree.

Please note that an item cannot be more than one level deeper than the preceding item. Also, while the initial level is arbitrary, no item may ever be less deep than the first item.

```
levels←1 2 2 1
items←'item'∘,¨⍕¨⍳⍴levels
html←req #.JQO.treeview 'tree' levels items
```





11.3 APLJax: Changing Page Content Without A Load

In the early days of the web, each page request would get all the HTML to build the page each time. If a page had to change depending on user input, the client needed to submit a form and rebuild the whole page. Now, client side languages can communicate with the server outside a form submission and can update parts of a page without reloading the whole thing. The combination of web technologies that gets this done is called Asyncronous JavaScript And XML, or AJAX.

The APLJax Method

AJAX allows the server to handle processing and update page content in between whole page loads. This is achieved through a slightly different kind of HTTP request, called an XMLHTTP request. MiServer identifies these requests and updates req.IsAPLJax, a Boolean indicating whether it is or is not an AJAX request. If this is true, MiServer will then attempt to call the method APLJax in your MiPage.

APLJax is constructed just like the Render method. It is:

- monadic
- public
- passed the request object

```
 APLJax req
:Access Public
 Your Response
```

If this method is called, req.NoWrap becomes true, and the HTML passed to req.Return does not have the template applied to it.

If this method does not exist, Render will be called as normal. You could consider a control structure using req.IsAPLJax if you decide to not use the APLJax method. Our examples will explore both choices.

Currently Implemented AJAX Interactions

We have built the first few functions which bring this capability to MiServer. Eventually, we plan on having a suite of tools, much like HTMLInput which help you to integrate AJAX into your MiPages, which we will call APLJax. For now, we have a few examples to put you in the right direction for building your own.

The Drag and Drop Interaction

jQueryUI also includes a function which allow HTML elements to be clicked and dragged across the screen, as well as a function which allows elements to respond to having a ‘draggable’ element dropped on them, for which JQUI.Draggable and JQUI.Droppable are cover functions. In our first example, we will create a MiPage that uses AJAX to dynamically update some HTML in response to a draggable element being dropped on a droppable element.

```
:Class DragDropAPLJax : MiPage

    :Include #.HTMLInput

    :Field Public drag←'No Selection'


Render req;html;Drags;Drops;output;script;dragsel;dropid;outputsel
    :Access Public

    :If ~req.IsAPLJax
```

req.IsAPLJax is 1 if the HTTP request is an AJAX request.

```
      Drags←'h1 id="1" class="drag"' Enclose 'Drag - 1'
      Drags,←'h1 id="2" class="drag"' Enclose 'Drag - 2'
      Drops←'h2 id="drop"' Enclose 'Drop On Me'
      Output←⊃Enclose / 'div id="output"' 'h1' drag
```

If it is a normal request, build the HTML for the page.

```
      dragsel←'.drag'
      Script←req #.JQUI.Draggable dragsel
```

Build the script for the draggable elements with JQUI.Draggable (See Appendix B for a complete reference). This function:

- Applies jQueryUI’s draggable behavior to a specified HTML element or elements, enabling them to be moved by a mouse click and drag
- Clones the draggable element during the drag, which disappears when dropped

```
      dropid←'drop'
      outputsel←'#output'
      Script,←req #.JQUI.Droppable dropid dragsel outputsel
```

Build the script for the droppable elements with JQUI.Droppable (See Appendix B for a complete reference). This function:

- Applies jQueryUI’s droppable behavior to a specified HTML element or elements, allowing them to respond to draggable elements being ‘dropped’ on them.
- The function will only react to the elements which include the selector listed in its second argument
- During a drop, an AJAX request is cued. The id of the dropped element is passed back to the server in the body of the request as the value of the name ‘drag.’ It is interpreted by

MiServer as form-data, and is treated as such, populating req.Data and similarly named public fields.

```
      html←Drags,Drops,Output,Script
    :Else
      html←'h1' Enclose 'You dragged ',drag
      req.Response.NoWrap←1
```

If the request is an AJAX request, send back HTML to be placed in tag. Since this is not the HTML for the whole page, we set req.Response.NoWrap to 1 so the server will not wrap it in the page template. When the browser receives the HTML, it will update the tag.

```
    :EndIf
    req.Return html



:EndClass
```

Sortable Lists

JQUI.Sortable allows users to click and drag list items, reordering them or even moving them between several lists. If set to callback to the server, it reports the new indices with an AJAX request.

```
list1←'To Do' 'Wash Car' 'Vacuum' 'Laundry' 'Cut Grass' 'Clean Garage'
list2←'Done' 'Go Fishing'
useHdrs←1
listIDs←'s1' 's2'
listContent ← list1 list2
jqpars←''
chain←''
callback←1
html←req #.JQ.Sortable useHdrs listIDs listContent jqpars chain callback
```

Below are the fields populated by the serialized data. In this example, the third element of list s1 is now the second element of list s2:

```
    s1
s1[]=1&s1[]=2&s1[]=4&s1[]=5
    s2
s2[]=1&s1[]=3
```

11.5A More General AJAX Solution - JQ.On

For a more general solution for working with AJAX, we present JQ.On, a cover for the JQuery function .on( ). On allows you to identify an event, like a keypress or a mouse click on an element, which triggers an AJAX request. This request may contain data and the response may update a page element or execute jQuery functions.

To illustrate this, we will build an example which when a button is pressed, takes the text from a text edit field, and then displays that text by placing it within another element in the body.

First, we build the MiPage with the HTML infrastructure. It contains a button, a text field and a containing the text 'Start'.

```
:Class OnExample : MiPage
:Include #.HTMLInput

:Field Public Text←'Start'


 Render req
   :Access Public
   html←BRA 'Text' Edit Text
   html,←BRA 'button' Button 'Click Me'
   html,←'div id="display"' Enclose Text
   ⍝ Where JQ.On code will go
   req.Return html


:EndClass
```

At this point the page is inert. There is no form to facilitate communication with the server. This will be the job of JQ.On, which will establish the condition that initiates a request, what data is sent back and what page element will be changed by the response.

When the button is clicked, it will send the value of the text field to the server. Once the text is processed, it will then update the contents of the with the attribute id="display".

```
⍝ Where JQ.On code will go
selector←'#button' ⍝ When the button...
event←'click'  ⍝ is clicked...
data←('Text' '#Text' 'attr' 'value') ⍝ get text from textbox...
update←'#display' ⍝ and update display with the response
html,←req JQ.On selector event data update
```

When this request is triggered, it will send up to three name value pairs in its body:

- event – the event specified in the second parameter
- what – if the element which triggers the request has an id attribute, that id is passed as the value of this name. If one does not exist, what is not passed.

- The variable set in the first parameter of the data parameter – in this case Text – this will be the name of the data generated by the data parameter

Keep these in mind to avoid name conflicts.

Notice that we have already included a field to handle the data passed in the Text variable.

Finally we need to respond to the AJAX request with the APLJax method.

```
∇ APLJax req
  :Access Public
```

We generate some HTML (or JSON as described later)…

```
  html←'p' Enclose Text
```

…and send the response

```
  req.Return html
```

The Completed Example

```
:Class OnExample : MiPage
:Include #.HTMLInput

:Field Public Text←''


 Render req;html;selector;event;data;update
   :Access Public
   html←BRA 'Text' Edit Text ⍝ Build the HTML
   html,←BRA 'button' Button 'Click Me'
   html,←'div id="display"' Enclose Text

   ⍝ Begin building JQ.On
   selector←'#button' ⍝ When the button
   event←'click'  ⍝ is pushed
   data←('Text' '#Text' 'attr' 'value') ⍝ get text from textbox
   update←'#display' ⍝ and update #display with the response
   html,←req JQ.On selector event data update
   req.Return html



∇ APLJax req;html
   :Access Public
   html←'p' Enclose Text ⍝ wrap the text in <p> tags
   req.Return html


:EndClass
```

Building the JQ.On step by step

On identifies a page element or elements which will generate an XMLHTTP request when a particular jQuery event occurs. That request can contain data and can update a page element or elements. The following section will examine each parameter.

JQ.On takes parameters as such:

{req} JQ.On selector event data update
where data is variable selector type parameter

Select the element(s) to bind the event

The first parameter, selector, determines which element or elements on your webpage . There are two options:

- Direct events – events which are directly bound to elements
  - Specified by a character vector, denoting the selector. The selector must be preceded by the appropriate symbol.
  - These events are only bound to HTML elements that are present when the page loads, meaning that dynamically generated HTML content will not trigger direct events
  - More efficient for events that will be bound to one or a few elements
- Delegated events – events which are bound to a parent element which monitor its children rather than attaching an event to every child. When the appropriate event is performed by a selected child, the script is 'delegated' to that child
  - Specified by a two element vector of selectors, the first being the containing HTML element, the second being the element inside of it that will trigger the event
  - Supports dynamic content. The HTML element which triggers a delegated event can be created after the page has been loaded.
  - This is more efficient when there are many page items which have the same event associated with them.

Select the event to be bound

The second parameter, event, is a character vector specifying the JQuery event object which will trigger the request. There are a quite a few predefined events, which cover just about every conceivable interaction a user can have with a web page. A more complete listing can be found on the jQuery website at: `http://api.jquery.com/category/events/`

Several of these are listed below.

Event Name	Description
click	The mouse is clicked
dblclick	The mouse is double clicked
hover	The cursor hovers over the elements space on the page
keydown	A key is pushed down
keypress	A key is pushed and it comes up

keyup	A key comes up
submit	A submit action is initiated
select	A selectable element (such as a text edit field) is selected
scroll	An item that can be scrolled is scrolled
resize	An element is resized
mousedown	A mouse button is depressed
mouseenter	The cursor enters the space occupied by an elemnt
mouseleave	The cursor leaves the space occupied by an element
mousemove	The cursor moves within the space occupied by an element
mouseup	The mouse button is released

It is also possible to implement your own events, but that is outside the scope of this chapter.

Gather data

The third parameter, `data`, is either empty to return no data or a vector of vectors which specifies the data from the page to be sent as a name value pair in the body of the request. There are several sources of data, from the HTML contained by an element, to an element's attributes to CSS properties. We use five different commands, outlined below, to select different the types. This makes the syntax of this parameter a bit more complicated, as different types may or may not require an additional parameter. In any case, the general format of `data` is:

- If the data is being collected from the element which initiates the request

```
data ← variable type {parameter}
```

- If the data is being collected from a different element

```
data ← variable selector type {parameter}
```

Element	
variable	the name that the data will be passed back as the value of, to be available to public fields of the same name and req.Data
selector	the element from which the data will come. It must be a single element. As noted above, if this is not specified the data will be collected from the element that has initiated the request
type	the type of data to be acquired, described below
parameter	some types of data require parameters as described below

Type	What will be returned	Parameter
html	all the HTML contained by the element	none
serialize	any data which can be serialized in the element	none

css	returns the value of a css property	css property
attr	returns the value of an HTML attribute – can be a non-standard attribute	html attribute
is	tests the element against a JQuery selector and returns true or false	JQuery selector, see note below

JQuery selectors and .is()

JQ.On incorporates the `.is()` function, which can test whether elements would be selected by a particular selector. For example, we have a number of elements with a particular class and would like to know if the element we have selected is a list item.

```
html,←#.JQ.On '.class' 'click' ('data' 'is' 'li') ''
```

When an element with the class '`class`' is clicked, it send a request to the server containing the name value pair "`data=true`" or "`data=false`" depending on whether the element is a list item.

Note that the parameters are selectors, but an expanded list from the selectors we learned about in Chapter 6. jQuery has introduced selectors that can test for certain states. For example `:contains('Frodo')` would select all elements which contain the text 'Frodo'. Note that this will not however select input elements which have the text string entered into them, as that text is actually the value of the value attribute of that tag.

While delving into the nuances of jQuery selectors is outside the scope of this text, you can learn about them here:

`http://api.jquery.com/category/selectors/`

Note: the selector `:checked` would do well in your tool box, as it returns true or false to the checked status of a checkbox or radio button.

Updating you page and a first look at JSON

Once you have received the request from your page, you may want to respond in such a way that it affects the look of the page. This could be as sweeping as changing the entire contents of the `<body>` tags or as subtle as adding some HTML to an existing element without disturbing the rest of its contents.

The last parameter, update, controls this. If a selector is specified in it, then the XMLHTTP response will replace the contents of the specified HTML element with the contents of the body of the request, the character vector passed to `req.Return`. Again, if `req.Response.NoWrap` is not true, the response will be wrapped in the template and likely you will not like the result. It is reset to false on every request.

If the parameter is left empty, we have a few more options. Instead of expecting HTML, script will be inserted in your MiPage which looks for JavaScript Object Notation (JSON) to pass to it one of a

number of commands. JSON is a data interchange format used to pass easily writeable data structures to JavaScript. Take a stop by http://www.json.org to learn more.

MiServer has the utility namespace #.JSON which contains a number of functions for converting data between APL, XML and JSON. While this is not yet documented here, expect that a future update will include a comprehensive guide to this namespace. For right now, we need only concern ourselves with #.JSON.fromNVP, which takes name value pairs and constructs JSON from them.

That said, there are four commands that can be passed back:

- execute – which immediate executes the JavaScript passed as its value

```
    #.JSON.fromNVP ('execute' JavaScript)
```

Note the format from the upcoming example:

```
Html←#.JSON.fromNVP('execute'('alert("You clicked ',what,'")'))
```

- replace – which replaces the content of an HTML element
- append – which adds new content to the end of an HTML element
- prepend – which adds new content to the front of an HTML element

These three share a syntax:

```
    command←'replace' ⍝ could be 'append' or 'prepend'
    selector←'#mydiv'
    content←'One does not simply walk into Mordor'
    #.JSON.fromNVP (command selector)('data' content)
{"data":"One does not simply walk into Mordor","replace":"#mydiv"}
```

An example implementation of a JSON controlled JQ.On

```
:Class JSonControlledJQON : MiPage
    ⎕ML←1
    :Include #.HTMLInput
    :field count←0
    :field public event
    :field public what


 Render req;html
      :Access Public
      html←'h2'Enclose'JQ.On testing'
      html,←2 BRA'mybutton'Button'Press Me'
      html,←'div id="mydiv"'Enclose'Starting content'
      html,←req #.JQ.On'#mybutton' 'click'
      req.Return html



 APLJax req;html
      :Access Public
      :Select 4|count
      :Case 0
        html←#.JSON.fromNVP('execute'('alert("You clicked ',what,'")'))
      :Case 1
        html←#.JSON.fromNVP('replace' '#mydiv')('data' 'This is my new content')
      :Case 2
        html←#.JSON.fromNVP('append' '#mydiv')('data' ' and This is my appended content')
      :Case 3
        html←#.JSON.fromNVP('prepend' '#mydiv')('data' 'This is my prepended content and ')
      :EndSelect
      count+←1
      req.Return html



:EndClass
```

11.6 The jqpars Attribute

Each jQuery function has a number of additional parameters which can be passed to them to modify their functionality. For the most part, the functions we have provided in JQ, JQUI and JQO represent the default functionality of the function they are covering, with some exceptions.

Each JQ* namespace function has been equipped with an argument called jqpars, which takes a character string of properly formatted JQuery parameters and places them in the appropriate section of the function call. Since there is no additional processing on this argument, the parameters must be passed as if they were going to be typed directly into the function.

We will follow the process of identifying and adding those parameters. For the purpose of this example, we will make a dialog box with modal behavior, disabling the other items on the page.

The example below is the function for a dialog box with the last parameter, jqpars, set to default with ''. On page load, the code produced by the below function will pop up a dialog box containing the text 'Some Content'.

```
    JQUI.Dialog 'objectid' 'Title' 'SomeContent' ''
<div id="objectid" title="Title">Some Content</div>
<script type="text/javascript">

/* <![CDATA[ */

$(function(){$("#objectid").dialog({});});

/* ]]> */

</script>
```

First we will look through the documentation on the function, found at `http://jqueryui.com/demos/dialog/` there is a list of the options.

There happens to be an option called 'modal' which adds the behavior we were looking for. It is described as a 'Boolean' with a default value of 'false.' Clicking it reveals a number of examples, including the following:

```
Code examples

Initialize a dialog with the modal option specified.
$( ".selector" ).dialog({ modal: true });
```

So by setting the fourth parameter, jqpars, to 'modal:true' the dialog box displays modally.

```
    JQUI.Dialog 'objectid' 'Title' 'SomeContent' 'modal:true'
<div id="objectid" title="Title">SomeContent</div>
<script type="text/javascript">

/* <![CDATA[ */
```

```
$(function(){$("#objectid").dialog({modal:true});});

/* ]]> */

</script>
```

There are several additional parameters for each function in the JQ* namespaces. Feel free to take a look through the documentation for the jQuery calls themselves, links to which are included in the function's reference in Appendix B.

Appendix A: HTML and HTMLInput

The following workspaces are designed to help you build HTML in MiPages with some APL-like syntax.

A.1 HTML Namespace Function Reference

HTML.*
Insert HTML Tags

html ← {attrs} HTML.***fn*** innerhtml

attrs The optional left argument contains any additional attributes for the HTML tags. These can be passed as either an N × 2 matrix of attribute-value pairs, a character vector or as a vector of vectors where each element contains two character vectors representing the name and value and a vector of vectors of depth 2 of alternating name values

fn The function which produces a tag of the same name

a	div	h1	head	ul	p
b	font	h2	html	li	pre
body	form	h3	input	link	span

The currently implemented functions are:

attributes Any additional attributes to be placed in the opening tag (default is '')

```
    HTML.h1 'Title'
<h1>Title</h1>

    'type' 'post' HTML.form 'Things in the Form'
<form type="post">Things in the Form</form>

    'id' 'bat' 'class' 'flying'HTML.div 'content'
<div id="bat" class="flying">content</div>

    (2 2⍴'id' 'bat' 'class' 'flying')HTML.div'content'
```

```
<div id="bat" class="flying">content</div>
```

A.2 HTMLInput Namespace Function Reference

Background

The HTMLInput namespace contains a number of more complex functions that create HTML. Originally, this namespace was designed to make working with the dynamic functionality of the `<input>` tag more APL-like, but has since expanded to include a number of other functions.

HTMLInput.APLToHTML Convert APL Code to HTML

HTMLInput.APLToHTML preserves the formatting of APL code passed to it. It inserts space 'exit symbols' into every space, wraps the code in `<pre>` tags to preserve character turns, and sets the font to APL385 Unicode.

html←**APLToHTML** apl

apl A character vector of APL code

```
    ⎕vr 'foo'
    ∇ r←foo goo
[1]    r←1
[2]    :If goo
[3]        r←2
[4]    :EndIf
    ∇

    HTMLInput.APLToHTML ⎕vr 'foo'
<pre style="font-family:APL385 Unicode">    
           ∇ r←foo goo
[1]    r←1
[2]    :If goo
[3]        r←2
[4]    :EndIf
     ∇</pre>
```

HTMLInput.BRA Insert
 Tag after HTML

HTMLInput.BRA concatenates one or more `<br />` tags to the end of the right argument. `<br />` tags are read by browsers as line breaks.

html ← {n} **BRA** html

n	The number of tags to insert (default is1)
html	HTML to insert break tags after

```
    'x',(HTMLInput.BRA'y'),'z'
xy<br />

z
```

HTMLInput.BR Insert
 Tag Before HTML

HTMLInput.BR concatenates one or more `<br />` tags to the beginning of the right argument. `<br />` tags are read by browsers as line breaks.

html ← {n} **BR** html

n	The number of break tags to insert (default is1)
html	HTML to insert break tags before

```
    'x',(HTMLInput.BR'y'),'z'
x<br />

yz
```

HTMLInput.Button HTML Button

HTMLInput.Button generates an `<input>` tag with the `type="button"` attribute. This renders as a rectangular pushable button. This element has no function on its own and requires additional scripting to make use of it.

html ← name **Button** pars

where pars is value {attributes}

name	The value of the name attribute
value	The value of the value attribute, displayed as text on the button
attributes	Any additional attributes to be placed in the tag. (default is '')

```
    'Btn' HTMLInput.Button 'Click Me'
<input  type="button"  id="Btn" name="Btn" value="Click Me" />
```

HTMLInput.Checkbox HTML Checkbox

HTMLInput.Checkbox generates an <input> tag with the type="checkbox" attribute. This renders as a square, selectable box.

html ← name **Checkbox** pars

where pars is {checked} {attributes}

name	The value of the name and id attributes
checked	This sets the checked status of the checkbox (default 0). If 1, the checkbox is checked
attributes	Any additional attributes to be placed in the opening tag (default is '')

```
    'checkId' HTMLInput.CheckBox 0
<input  type="checkbox" id="checkId" name="checkId" />


    'checkId' HTMLInput.CheckBox 1
<input  type="checkbox" id="checkId" name="checkId" checked="1" />
```

HTMLInput.DropDown HTML Dropdown Menu

HTMLInput.DropDown generates a <select> tag that wraps a number of <option> tags, which in turn wrap character vectors. This renders as textbox with a button on it that lists all of the options. The textbox will auto-fill with the vectors contained in the options as dictionary elements.

html ← name **DropDown** pars

where pars is items {value} {attributes} {sort}

name	The value of the name and id attributes.
items	An n element vector of the items to be selected from in the dropdown menu. (default is 'Item1' 'Item2')
value	The value to be displayed when the dropdown box is generated. (default is 'Item1')
attributes	Any additional attributes to be placed in the opening tag. (default is '')
sort	1 or 0 (the default) If 1, and value matches one element of the items, will place the selected tag within the item and move it to the front of the list

```
      items←'Item 1' 'Item 2' 'Item 3'
      'DDown' HTMLInput.DropDown items 'Item 2' '' 1
<select  id="DDown" name="DDown">

<option value="Item 2" selected="selected">Item 2</option>

<option value="Item 1">Item 1</option>

<option value="Item 3">Item 3</option>

</select>
```

HTMLInput.Edit
HTML Text Field

HTMLInput.Edit generates an <input> tag with the type="text" attribute. This renders as an editable text box, similar to the Edit object of the Dyalog APL GUI.

html ← name **Edit** pars

where pars is {value} {size} {attributes}

name	The value of the name and id attributes

value	The text to be displayed in the element (default is '')
size	The maximum character count of the text box (default is 0)
attributes	Any additional attributes to be placed in the opening tag (default is '')

```
    'theId' HTMLInput.Edit 'SomeText' 20
<input  type="text" size="20" id="theId" name="theId" value="SomeText" />
```

HTMLInput.Enclose Wrap HTML with Tag

HTMLInput.Enclose wraps the right argument with HTML tags defined by the left argument.

html ← attribute **Enclose** innerHTML

=

attribute	The tag that will be enclosing the HTML. This can include any number of additional attributes
innerHTML	A character vector of HTML to be wrapped by the tags

```
    'h1' HTMLInput.Enclose 'A Title'
<h1>A Title</h1>


    'h1 id="tree"' HTMLInput.Enclose 'A Title'
<h1 id="tree">A Title</h1>
```

HTMLInput.File HTML File Upload Button

HTMLInput.File generates a file type `<input>` tag. This tag is generates a button with the text 'Upload Files' on it. When pressed, a file browser appears and a file can be selected. Next to the button is text that either says 'No File Selected' or the name of the selected file.

When a tag is submitted with a binary file, it returns a two element result, the name of the file and the data of the file.

Filename data ← name **File** pars

where pars is size {value} {attributes}

name	The value of the name attribute and the id attribute
size	The maximum character count of the file box
value	A character string that will be displayed in the text field
attributes	Any additional attributes to be placed in the opening tag (default is '')

```
    'Upload' HTMLInput.File '40'
<input  type="file" size="" id="Upload" name="Upload" value="40" />
```

Note:
This tag is not evenly supported by all browsers.

Usage Example:

```
:Class Upload : MildPage

   :Include #.HTMLInput

   :Field Public Action←''          ⍝ Action button
   :Field Public Upload←''          ⍝ File Upload?

   ∇ Render req;html
     :Access Public
     DoAction
     html←'Upload' File '40'
     html,←'Action' Submit 'Upload'
     html←req('post'Form)html
     req.Return html
   ∇

   ∇ DoAction;ftype
     :If 0≠⍴Upload
         file data←Upload
     :AndIf 0≠⍴file←(1-⌊/(⍳⍴file)×'\/')↑file
         ftype←⍕ (#.Strings.lc ¯4↑file)
         (tn file)← ftype #.Files.CreateTemp req.Server.TempFolder
         data ⍪NAPPEND tn
         ⍪NUNTIE tn
   :EndIf
   ∇


:EndClass
```

HTMLInput.Form Insert HTML Form

HTMLInput.Form wraps a section of HTML in `<form>` tags. Form tags are used in conjunction with a submit action to generate an HTTP request for a resource. These tags wrap a portion of an HTML document that often contains elements that can return name/value pairs to the server, which can be used to additionally process the request.

`<form>` tags signify the data that will be contained in the leading line of the HTTP request, including the type of HTTP request, what resource will be requested and which HTTP method will be used.

html ← {atts} (method **Form**) innerHTML

method	The type of HTTP request to be sent. A form can only use the GET or POST HTTP request types
atts	Any additional attributes to be placed in the opening tag (default is '')
innerHTML	A character string of the HTML to be wrapped by the tag

```
    ...
    html←'Text' HTMLInput.Edit Text
    html,←'Action'Submit'Submit'
    html←req('post'Form)html
    ...

<form action="/reverse.dyalog" method="post" enctype="multipart/form-
        data">
<h2>Reverse Text Example</h2>
<br/>
Enter Text:
<input type="text" size="" id="Name" name="Name" value="" />
<br/><br/>
<input type="submit" id="Action" name="Action" value="Reverse" />
<input type="submit" id="Action" name="Action" value="Clear" />
</form>
```

Note:
When using the "POST" method, HTMLInput.Form adds '`enctype="multipart/form-data"`' to the leading form tag.

HTMLInput.Hidden HTML Hidden Field

HTMLInput.Hidden inserts a hidden type <input> tag into your MiPage. These tags are useful for storing information that you do not want the user to see between requests. One common technique is to use hidden type <input> tags is to preserve information between requests.

html ← name **Hidden** pars

where pars is {value} {attributes}

name	The value of the name and id attributes
value	The value(s) of the value attribute
attributes	Any additional attributes to be placed in the opening tag (default is '')

```
    'HideMe' HTMLInput.Hidden 'Data the user cannot see'
<input  type="hidden" id="HideMe" name="HideMe" value="Data the user
        cannot see" />
```

HTMLInput.JS Insert JavaScript

HTMLInput.JS wraps the right argument with tags and JavaScript syntax as if it was a line of JavaScript.

html ← **JS** script

script	A character vector of JavaScript

```
    jscript←'document.write("<h1>This is a heading</h1>");'
    HTMLInput.JS jscript
<script type="text/javascript">

/* <![CDATA[ */

document.write("<h1>This is a heading</h1>");

/* ]]> */

</script>
```

HTMLInput.List Ordered and Unordered Lists

HTMLInput.List creates ordered (numbered) or unordered (bulleted) lists. The function wraps list items in `<li>` tags and all of the items with `<ul>` or `<ol>` tags.

html ← {name} **List** pars

where pars is items {ordered}

name	A value of the id attribute of the `<ul>` or `<ol>` tag
items	A vector of list items
ordered	0, the default, or 1 – If 1, the list will be ordered

```
    'List' HTMLInput.List 'apple' 'ball' 'cactus'
<ul id="List"><li>apple</li>
<li>ball</li>
<li>cactus</li>
</ul>

    'List' HTMLInput.List ('apple' 'ball' 'cactus') 1
<ol id="List"><li>apple</li>
<li>ball</li>
<li>cactus</li>
</ol>
```

HTMLInput.MultiEdit HTML Multiple Row Text Field

HTMLInput.MultiEdit creates a multi-row/column text editing field. This is returned to the server as a character vector with preserved character turns. In that sense, it must be modified or enclosed in `<pre>` tags to maintain formatting.

html ← name **MultiEdit** pars

where pars is (rows cols) {values} {attributes}

name	The value of the name and id attributes
rows	Rows of text field, in characters (default is 10)
cols	Columns of text field, in characters (default is 40)
value	Text displayed in the text field (default is '')
attributes	Any additional attributes to be placed in the tag (default is '')

```
    'TArea'  HTMLInput.MultiEdit (10 10) 'Content'
<textarea  id="TArea" name="TArea" rows="10" cols="10">
```

```
Content
</textarea>
```

HTMLInput.Password HTML Password Field

HTMLInput.Password creates a password type `<input>` tag. It is similar to the text type `<input>` tag, except that it displays only hashes to the user. This is ideal for entering passwords, as its name suggests.

html ← name **Password** pars

where pars is {size} {value} {attributes}

name	The value of the name and id attributes
size	The maximum character count of the password box
value	The text value of the password box (default is '')
attributes	Any additional attributes to be placed in the opening tag (default is '')

```
    'theId' HTMLInput.Password 'Y0urP4ssw0rd' 20
<input  type="password" size="20" id="theId" name="theId"
        value="Y0urP4ssw0rd" />
```

Note:
Be aware that the data sent by this input type is not encoded by default, and sending the information via a GET HTTP request will display the password in the URL bar.

HTMLInput.RadioButton HTML Radio Button

HTMLInput.Radio inserts a radio type `<input>` tag, which produces a radio button.

html ← name **RadioButton** pars

where pars is {checked} {value} {attributes}

name	the value of the name and id attributes

checked	0, the default, or 1 if 1, the button is checked
value	the value of the value attribute (default is name)
attributes	any additional attributes to be placed in the opening tag (default is '')

```
    'HideMe'HTMLInput.RadioButton ''
<input  type="radio" id="HideMe" name="HideMe" value="HideMe" />
```

Note:
Only one radio button can be selected by the user in each form. However, if multiple radio buttons are created with their checked attribute set to one and the user does not make a radio button selection before form submission, multiple radio buttons will pass a checked value.

HTMLInput.SP Insert Spaces Before HTML

HTMLInput.SP inserts encoded spaces into HTML that are preserved by the browser.

html←{n} **SP** html

n	the number of spaces to insert (default is1)
html	HTML to insert spaces before

```
    HTMLInput.SP 'Some HTML'
 Some HTML
```

HTMLInput.Submit HTML Submit Button

HTMLInput.Submit creates a submit type <input> tag. This renders as a button which, when placed inside <form> tags and clicked, initiates an HTTP request defined by the form.

html ← name **Submit** pars

where pars is value {attributes}

name	The value of the name attribute
value	The value of the value attribute, displayed as text on the button. Determines button size (default is 'Push Me!')
attributes	Any additional attributes to be placed in the tag (default is '')

```
   'Submit' HTMLInput.Submit 'Click Me'
<input type="submit"  id="Submit" name="Submit" value="Click Me" />
```

HTMLInput.Table Enclose Array in HTML Table

HTMLInput.Table formats vectors and matrices with a maximum depth of 2 with the tags associated with the <table> HTML elements.

html ← {name} **Table** pars

where pars is data {table_atts } {cell_attribs} {header_attribs} {header_rows} {cell_ids}

name	the value of the name and id attributes.
data	a matrix with no more than a depth of 2. (default is 'data')
table_atts	attributes to be placed in the leading ‘table’ tag (default is '')
cell_attribs	attributes to be placed in the cells of the table outside of the header. (default is '')
header_attribs	attributes to be placed in the table’s header rows. (default is '')
header_rows	number of rows that will be marked as the table’s header. (default is 0)
cell_ids	if 1 (default is 0), individual ids will be generated based on the indices of each cell in the style of r#c# - r for row.

Note: Example formatted for ease of reading

```
    data←(3 3⍴('hdr'∘,¨⊃,/⍳¨⍳3),'cell'∘,¨⊃,/⍳¨⍳6)
    HTMLInput.Table data '' '' '' 1 1

<table>
 <thead>
  <tr>
   <th>header1</th>
   <th>header2</th>
   <th>header3</th>
  </tr>
 </thead>
 <tr>
  <td id="r2c1">cell1</td>
  <td id="r2c2">cell2</td>
  <td id="r2c3">cell3</td>
 </tr>
 <tr>
  <td id="r3c1">cell4</td>
  <td id="r3c2">cell5</td>
  <td id="r3c3">cell6</td>
 </tr>
</table>
```

HTMLInput.TextToHTML Insert
 for Each CR

HTMLInput.TextToHTML preserves the lines of text passed to by replacing each character turn in the text with a `<br />` tag as well as inserting one at the end of the text.

html ← **TextToHTML** text

text	Character string of text

```
      text←'hello',(⎕UCS 13 10),'world'
      HTMLInput.TextToHTML text
hello<br/>

world<br/>
```

Appendix B: Base64 encoding

Base64 is an algorithm for converting binary data into ASCII strings for transfer via an HTTP Request. This encoding was originally designed as a way to encode emails, although server side languages may be able to encode data with this algorithm as well.

Functions which decode it are a common fixture of server languages, like PHP and ASP.Net.

B.1 Base64 Namespace Function Reference

Base64.Decode Encode Data as Base64

Base64.Decode takes a string of text which has been encoded with the Base64 algorithm and decodes it.

```
rc←Encode txt
```

txt	data which has been encoded with the Base64 algorithm

Note:
The output of this function is text representing the Unicode position of each 8bit section of the converted binary. As a result, if the original text contains characters which are 16 bit or higher, the output will not faithfully represent the encoded data. Instead, the text will need be additionally decoded like any normal byte stream of data in that character set.

For example:

When using the UTF-8 Charset, ρ2 is represented as `4o20Mg==`.

```
      Base64.Decode '4o20Mg=='
â ´2
      ⍝ the output does not match the original
      ⍝ convert to a byte stream
      ⎕UCS 'â ´2'
226 141 180 50
      ⍝ Since the UTF-8 Charset was used to encode the
      ⍝ original data, reencode the byte stream as UTF-8
      'UTF-8' ⎕UCS 226 141 180 50
ρ2
```

```
      'UTF-8' ⎕UCS ⎕UCS Base64.Decode '4o20Mg=='
ρ2
```

Base64.Encode Decode Base64 Encoded Data

Base64.Encode takes a character string and applies the Base64 encoding algorithm to that text. Currently, this function only supports text strings which contains characters with Unicode positions no higher than 256. Attempts to use unsupported characters will result in a TRANSLATION ERROR.

rc ← **Encode** txt

txt a character string

```
      Base64.Encode 'apple'
YXBwbGU=
      Base64.Encode 'ρ2'
TRANSLATION ERROR
      Base64.Encode'ρ2'
     ∧
```

Appendix C: JQ, JQUI, and JQO

Background

jQuery is a JavaScript library which contains many commonly used scripts. JQ, JQUI and JQO are a namespaces designed to simplify jQuery integration into your MiPages.

- JQ – interfaces with the jQuery library
- JQUI – interfaces with jQueryUI plugins
- JQO – interfaces with third party jQuery plugins

These three namespaces have implemented only a few of the vast menagerie of jQuery plugins and widgets, but can be considered an example of how to integrate jQuery functionality in an APL environment.

Location of Scripts and style sheets

With the exception of JQ.JQueryfn, each of the following functions calls a function from a jQuery library or plugin. Each of the functions links the appropriate resources to the HTML page with the resource mapping feature. The resources for all of the included plugins can be found in `ServerRoot/Plugins/`.

If you need to modify the location of the scripts and style sheets, refer to the Resource Mapping and Virtual Directories references in Appendix F.

Note:
Each of the following functions that have a user interface component has an example of implementation and a screen shot in Section 10.2.

C.1 JQ Namespace Function Reference

Currently, this namespace contains a single pertinent function. JQueryfn is used in every other function to build the script:

JQ.JQueryfn
Build JQuery call

JQ.JQueryfn generates a JQuery call wrapped in <script type="text/javascript"> tags and an XML CDATA section, which marks the function as not-to-be-parsed character data.

html ← **JQueryfn** pars

where pars is JQueryFunName HTMLsel {JQueryFunPars} {JQueryFunChain}

JQueryFunName	The name of the JQuery function (default is '')
HTMLsel	The selector of the HTML element or elements affected by the JQuery widget (default is '') Note the following syntax: • Character strings will have a '#' appended to the beginning of the string, identifying the first element as the value of an 'id' attribute • Character strings enclosed in double quotes will be passed without appending anything to their front
JQueryFunPars	The parameters passed to the JQuery function (default is '')
JQueryFunChain	The code for any chained functions (default is '')

```
JQ.JQueryfn 'tablesorter' 'table' '' ''
<script type="text/javascript">
/* <![CDATA[ */
$(function(){$("#table").tablesorter({});});
/* ]]> */
</script>
```

JQ.On
Binds JQuery Event

JQ.On binds a JQuery event handler to an element and gives the user a means to respond to that event via AJAX requests. While this function is described below, its implementation contains complexities not described below. You are encouraged to read section 10.6, which takes an in depth look at this function.

html ← {req} **On** pars

where pars is selector events data update
where data is dvariable dselector dtype dparameter

Note: If a target is not specified by the parameter, update is added to the JavaScript can respond to a page which JSON few specific There are four structures. commands: supported

req	the request object. Required for compatibility with Internet Explorer
selector	the element or elements on which the event will be bound. This parameter is either a character vector, to specify a direct binding or a two element vector specifying a delegated binding.
event	the jQuery event which will trigger the request. A list of valid events can be found here:
data	either empty passing no data or a vector of parameters as described above and detailed below.
update	either a selector, signifying the HTML element which will have its contents replaced by the body of the response, or empty which sets up a handler to take a JSON structure (see note below for syntax)

Parameters of Data	Description
dvariable	the name that the data will be passed back as the value of, to be available to public fields of the same name and req.Data
dselector	the element from which the data will come. It must be a single element. As noted above, if this is not specified the data will be collected from the element that has initiated the request
dtype	the type of data to be acquired, described below
dparameter	some types of data require parameters as described below

Type	What will be returned	Parameter
html	all the HTML contained by the element	none
serialize	any data which can be serialized in the element	none
css	returns the value of a css property	css property
attr	returns the value of an HTML attribute – can be a non-standard attribute	html attribute
is	tests the element against a JQuery selector and returns true or false	JQuery selector

- execute – which executes some JavaScript

```
    script←'alert("You clicked the button")'
    #.JSON.fromNVP('execute' script)
{"execute":"alert(\"You clicked the button\")"}
```

- replace – replaces the contents of an HTML element
- append – adds to the end of an HTML element
- prepend – adds to the beginning of an HTML element

```
    command←'replace'  could be 'append' or 'prepend'
    selector←'#mydiv'
    content←'One does not simply walk into Mordor'
    #.JSON.fromNVP (command selector)('data' content)
{"data":"One does not simply walk into Mordor","replace":"#mydiv"}
```

C.2 JQUI Namespace Function Reference

The following functions use the jQuery library and official jQueryUI plugins. Chapter 10 contains simple examples for each of the following and screen shots of the widgets in action.

JQUI.Accordion jQuery Accordion Widget

JQ.Accordion renders as an object with a number of collapsible content blocks that when collapsed show only their headings. It creates a containers that contains a number of containers each preceded by a header element.

Additional parameters can be found at `http://jqueryui.com/demos/ accordion/`.

html ← {req} **Accordion** pars

where pars is id {hdrs} {content} {jqpars}

req	The HTTPRequest object
id	The id attribute for the Accordion
hdrs	An n-element array of header names for each Accordion folder
content	An n-element array of content for each accordion folder
jqpars	Additional Accordion JQuery parameters

JQUI.DatePicker jQuery DatePicker Widget

JQ.DatePicker creates a text type input box which, when clicked, pops up a dialog containing a calendar. A date selected from this is entered into the input box.

Additional parameters can be found at `http://jqueryui.com/demos/ datepicker/`.

html ← {req} **DatePicker** pars

where pars is id {editpars} {jqpars}

req	the HTTPRequest object
id	the id for the DatePicker

editpars	the Parameters for the text field (see HTMLInput.Edit)
jqpars	additional DatePicker parameters

JQUI.Dialog jQuery Dialog Widget

JQ.Dialog creates a pop up within the browser that resembles a system dialog. By default, this box has a title and can hold any HTML contents. It is draggable, resizable and pops up when the screen is loaded.

Additional parameters can be found at `http://jqueryui.com/demos/dialog/`.

html ← {req} **Dialog** pars

where pars is id {title} {innerHTML} {jqpars}

req	The HTTPRequest object
id	The id attribute of the Dialog
title	The title for the Dialog window
innerHTML	The HTML displayed in the body of the Dialog window
jqpars	additional dialog parameters

JQUI.Draggable jQueryUI Dialog Widget

JQ.Draggable identifies the element or elements in your HTML document which can be clicked and dragged across the screen. While the default parameters of the jQueryUI function allow the widget itself to be dragged across the screen, we have applied the ‘clone’ parameter to it. This will spawn a clone of the draggble object that will be moved around with the mouse and disappear when the mouse button is released. These parameters can be discarded by making any changes to the jqpars argument.

Additional parameters can be found at `http://jqueryui.com/demos/draggable/`.

html ← {req} **Draggable** pars

where pars is id {jqpars}

req	the HTTPRequest object
id	the selector(s) (generally ids) for the item to be dragged
jqpars	additional Draggable parameters

JQUI.Droppable jQueryUI Droppable Widget

JQ.Droppable identifies an element of your HTML page that will respond to droppable objects being dragged on top of them and released.

In this implementation of Droppable, selectors are specified to indicated which droppable elements will trigger a response. When one such element is dropped on top of the droppable element, an AJAX request is sent to the server contain the name-value pair Drag-(the id of the dropped element) in the Data header .

Additional parameters can be found at `http://jqueryui.com/demos/droppable/`.

html ← {req} **Droppable** pars

where pars is id accept {update} {jqpars}

req	the HTTPRequest object
id	the id attribute of the Droppable element
accept	the selector(s) which identify the Draggable elements which will affect this particular Droppable
update	the selector(s) of the elements which will be updated by the response to the request triggered by a drop. If empty, no element will be updated (default is '')
jqpars	additional Droppable parameters. Any input will override the current parameters, which include those responsible for the callback and the update

JQUI.Sortable Sortable List Widget

JQ.Sortable renders one or more vectors of items as a number of lists which have list items that can be dragged to different positions within their list and between different lists contained by the generated container element.

This implementation of Sortable includes an option to post the new list positions back to the server with an AJAX request.

Additional parameters can be found at `http://jqueryui.com/demos/sortable/`.

html ← {req} **Sortable** pars

where pars is usehd ids lists {styles} {jqpars} {chain} {callback}

req	the HTTPRequest object
usehd	0 or 1 – If true, the first item in each list is no longer sortable, for use as a header
ids	a vector of ids, one for each list
lists	a list of items or a vector of lists of items. Can contain HTML
styles	a two element vector of vectors of name value pairs adding styles to the entire list and the list items respectively.
jqpars	additional Sortable parameters
chain	any jQuery to be chained onto the call
callback	0 or 1 – If true sends an AJAX request back to the server with the serialized data from the list.

Note:
The callback parameter will return an XMLHTTP request with the updated indices every time the lists are modified.

Below is the data in public fields named after the serialized lists. In the below example, the third element of list s1 is now the first element of list s2:

```
    s1
s1[]=1&s1[]=2&s1[]=4&s1[]=5
    s2
s1[]=3&s2[]=1
```

JQUI.Tabs
jQuery Tab Widget

JQ.Tabs renders as an object with a list of selectable titles across the top. When a title is clicked, its associated content comes into focus.

Additional parameters can be found at `http://jqueryui.com/demos/tabs/`.

html ← {req} **Tabs** (id tabnames content jqpars)

req	the HTTPRequest object
id	the id attribute of the tabs
tabnames	n-element vector of charvec of names to appear on the tabe
content	n-element vector of charvecs with the HTML content for each tab or a URI that specifies which file in the server is dynamically loaded as the tabs content when the tab is active
jqpars	additional Tabs parameters

Note:
A tab's content can either be a character vector of HTML, or a URI directing to a local resource, such as 'HTML/file.html'.

C.3 JQO Namespace Function Reference

The following functions require third party plugins.

JQO.jsTree Tree View Widget Using jsTree Plugin

JQO.jsTree renders data as a collapsible tree.

Additional parameters for this can be found at `http://www.jstree.com/documentation.`

html ← {req} **jsTree** pars

where pars is id items levels {jqpars}

req	the HTTPRequest object
id	the id attribute of the list (default is '')
items	a vector of items. These can include HTML. (default is ⍬)
levels	a vector of numbers of the same length as items. These numbers represent the depth of the data and affect how the items are nested.
jqpars	additional jsTree parameters (default is '')

Note:
There are a few rules to properly format the levels argument:

- The first item is considered to be at the base level of nesting. The level number used to signify the base level can be any integer. No other items can have a lower level number than it
- The higher a number, the more nested the data. An item cannot be more than one level higher than the preceding item
- An item can be any number of levels lower than its preceding item down to the base level

JQO.TableSorter jQuery TableSorter Widget

JQO.TableSorter creates a sortable table with optional pagination. It is based on #.HTMLInput.Table.

Additional parameters for this can be found at `http://tablesorter.com/docs/.`

html ← {req} **TableSorter** pars

where pars is id tablepars {jqpars} {pager}

req	the HTTPRequest object
id	the id attribute of the table (default is '')
tablepars	the data and parameters for the table (see HTMLInput.Table) (default is ⍬)
jqpars	additional TableSorter parameters (default is '')
pager	0, the default, or 1 the use of the Pager plugin, which adds pagination to the table

JQO.TreeView Tree View Widget Using TreeView Plugin

JQO.TreeView renders data as a collapsible tree.

Additional parameters for this can be found at `http://bassistance.de/jquery-plugins/jquery-plugin-treeview/.`

html ← {req} **TreeView** pars

where pars is id items levels {jqpars}

req	the HTTPRequest object
id	the id attribute of the list (default is '')
items	a vector of items. These can include HTML. (default is ⍬)
levels	a vector of numbers of the same length as items. These numbers represent the depth of the data and affect how the items are nested.
jqpars	additional TreeView parameters (default is '')

Note:
There are a few rules to properly format the levels argument:

- The first item is considered to be at the base level of nesting. The level number used to signify the base level can be any integer. No other items can have a lower level number than it

- The higher a number, the more nested the data. An item cannot be more than one level higher than the preceding item
- An item can be any number of levels lower than its preceding item down to the base level

Appendix D: SQL

Background

SQL is a namespace designed to simply integrate SQAPL, Dyalog's ODBC compliant database interaction tool, with your MiSite. The following functions require a properly formatted Datasources.xml file, as described in Chapter 8.

D.1 SQL Namespace Function Reference

SQL.ConnectTo Connect to a Datasource

SQL.ConnectTo opens a connection with a datasource specified in `Datasources.xml`.

r ← **ConnectTo** database

database	the name the datasource, from the name element of one of the datasources described in `Datasources.xml`

```
    SQL.ConnectTo 'DoesNotExist'
601   Datasource "DoesNotExist" not found
```

Note:

- If the function opens a connection, it returns a two element vector (0 NameOfConnection).
- If it fails, it returns a three element vector, consisting of a return code, an empty vector and an error message.

SQL.Do Connect to and Query a Datasource

Similar to the Do function found in SQA, SQL.Do prepares, executes and returns the result of a SQL statement. Unlike SQA.Do, which is supplied a connection, SQL.Do is supplied a datasource name specified in Datasources.XML. It then queries the datasource and closes the connection.

r ← **Do** database sqlstmt [bindvars]

database	the name the datasource, from the name element of one of the datasources described in `Datasources.xml`
sqlstmt	the SQL statement to be executed
bindvars	data for bind variables, if any

Note:
SQL.Do returns a namespace containing three variables:

ReturnCode	0 if successful. An error code reference can be found in the SQAPL manual.
Data	the matrix of data returned
Columns	a vector of column names

Excerpt from sqldemo.dyalog

```
data←#.SQL.Do'ZipCodes' 'select * from ZipCodes where StateAbbr = :a<C2:
          order by Zipcode'state
```

Note:
SQL.Do, unlike SQA.Do, always fetches all of the data and has no block mode.

SQL.CloseAll Close All SQAPL Connections

SQL.CloseAll closes all SQAPL connections in your session. This is not recommended for use within MiPages, instead it is useful for server shutdown.

r ← **CloseAll**

Appendix E: HTTPRequest Reference

E.1 The Request Object

An instance of the HTTPRequest class is generated at each HTTP request. The request object has two purposes:

- To contain the HTTP request and parse it into a number of fields
- To contain the HTTP response as it is being built

The request object also contains a number of methods for querying information from the HTTP request and adding information to the HTTP response.

E.2 Parsing the HTTP Request

Each HTTP request is parsed and the information that it contains is distributed among the following fields:

Input	The request line of the HTTP request. This includes the type of request, the resource to be requested and the version of the HTTP being used to format the request
Headers	All the headers of the HTTP request
Command	The type of request (post or get)
Page	The name of the requested resource
Arguments	Any name-value pairs passed within the URL are stored in this field as a 2 × N matrix of name-value pairs
PeerCert	When using secure communications, the certificate presented by the client
Data	When a post request is encoded with data, the data gets stored in this field as a 2 × N matrix of name-value pairs
Cookies	A list of the cookies transmitted by the browser

E.3 Namespaces

HTTPRequest.Session Persistent Session Data

Notable Content

State	A namespace that persists between page loads in a session. Store session specific variables here

HTTPRequest.Server Stores Server Settings

Notable Content

Config	A namespace of variables generated from the elements of SiteRoot/Config/server.xml

HTTPRequest.Response Stores the HTTP Response

Notable Content

HTML	A variable that contains the all HTML, save that found in the <head> tag structure.
HTMLHead	A variable containing the HTML between the <head> tags
Status	The HTTP status code to be returned to the browser
StatusText	The HTTP status message to be returned to the browser
NoWrap	A Boolean value. If true, the content passed to req.Return will not be passed to MildPage.Wrap.

E.4 Functions

HTTPRequest.Return Set Response HTML/Header

HTTPRequest.Return sets req.Response.HTML to the right argument and appends the header-value pairs of the left argument to HTTPRequest.Response.Headers. This function is frequently used at the end of a MiPage's Render method.

{hdrs} **Return** html

hdrs	A character vector to be added to the HTTP headers in the response
html	The character vector of HTML

HTTPRequest.ReturnFile Set Response to Return a File

HTTPRequest.ReturnFile is called by MiServer when the request specifies a file that does not have the .dyalog extension. It flags HTTPRequest.Response.HTML to the right argument and appends the header-value pairs of the left argument to HTTPRequest.Response.Headers.

It also sets HTTPRequest.Response.File to 1, which cues MiServer to treat the contents of HTTPRequest.Response.HTML as the path of the file that

{hdrs} **ReturnFile** html

hdrs	A two element vector or N×2 array of name value pairs to be appended into the req.Response.Headers
html	The character vector of HTML to be appended to req.Response.HTML

Note: This function is called by MiServer when the request specifies a file that does not have the default extension (.dyalog).

HTTPRequest.GetHeader Retrieve Header Value

The HTTP request can be sent with any number of headers, each of which can be queried by HTTPRequest.GetHeader. If the right argument is the name of a header, the value of that header is returned.

value ← **GetHeader** header

header — A character vector representing a header in the HTTP request

HTTPRequest.GetCookie
Retrieve Cookie Value

A cookie on a client's machine will have its name and value passed in the HTTP Request header 'Cookies,' if the browser has allowed cookies and the cookie's path matches the path of the request. HTTPRequest.GetCookie returns the value of a cookie with a name that matches its right argument. If such a cookie does not exist, it returns an empty result.

value ← **GetCookie** name

name — The name of the cookie

Note:
Remember that the value of a cookie will not be available until it is returned by a new HTTP request.

HTTPRequest.SetCookie
Set a Cookie

HTTPRequest.SetCookie appends a set-cookie command to the HTTP response header, with the name, value, path and deletion date of the cookie. If the client's browser set to disallow cookies, this will not have an effect.

SetCookie {name value path keep}

name — The name of the cookie. (default is 'CookieName')

value — The value that will be passed with the cookie. (default is 'CookieValue')

path — The path with which the cookie will be associated. (default is '/')

keep — The number of days the cookie should remain on the client machine. (default is 30)

Note:
Cookies can be overridden without first deleting them.

For a comprehensive example, see section 7.2

HTTPRequest.DelCookie
Delete a Cookie

HTTPRequest.DelCookie sends a set-cookie request, setting the value of the cookie to nothing and setting its deletion date to days before the current date. This effectively deletes the cookie from the client machine.

DelCookie ctl

name	The name of the cookie. (default is 'CookieName')
path	The path with which the cookie is associated. (default is '/')

Note:
For a comprehensive example, see section 7.2

HTTPRequest.Script
Insert Script in <head> Tags

HTTPRequest.Script appends a <script> tag to HTTPRequest.Response.HTMLHead the contents of which are enlisted and enclosed in the page <head> tags.

{atts} **Script** x

atts	The HTML attributes to be placed in the script tag. If a type attribute is not included in the list of tags, 'type="text/javascript"' is inserted
x	The character vector of the script

```
'src="JQuery/jquery.js"' aa.Script ''
      aa.Response.HTMLHead
<script src="JQuery/jquery.js" type="text/javascript"></script>
```

HTTPRequest.Style
Insert CSS Link

HTTPRequest.Style associates a style sheet with your page. It creates a <link> tag with the rel="stylesheet" and type="text/css" attributes and places the style file location in an href="" attribute. This is appended to HTTPRequest.Response.HTMLHead, which is the content placed between the <head> tags of the page.

Style file

file	The file path of a cascading style sheet

HTTPRequest.Title
Add a Page Title

HTTPRequest.Title wraps the supplied character vector in <title> tags and appends it to HTTPRequest.Response.HTMLHead the contents of which are enlisted and enclosed in the page <head> tags. This sets the title of the page that appears at the top of the browser.

Title x

x	A character vector

```
    aa.Title 'Example'
    aa.Response.HTMLHead
<title>Example</title>
```

HTTPRequest.Meta
Add a Page Title

HTTPRequest.Meta inserts meta tags into the <head> tags your page. Meta tags are used to categorize page content and provide keywords for indexing by search engines.

Meta attrs

attrs	Attributes for the meta tag

```
    aa.Meta 'name="description" content="Example"'
    aa.Response.HTMLHead
<meta name="description" content="Example"' />
```

HTTPRequest.Use Use a Resource Mapping

HTTPRequest.Use inserts `<script>` and `<style>` tags into the `<head>` tags of your page which are mapped to a name in the `Resources.xml` configuration file.

Use name

name	Name of a resource mapping defined in `Resources.xml`

Appendix F: Adding a JQuery Widget

Background

Adding a new JQuery widget to MiServer, involves integrating script files, possibly style sheets, and building the HTML required to support the widget. The JQ namespace contains a number of functions which support JQuery widgets. These functions only provide access to a small subset of what JQuery can do, but it is hoped that some of the more technically savvy MiServer users will take the trouble to learn how to add more JQ functions.

Adding support for a widget requires enough knowledge of JQuery and HTML to:

- Understand your JQuery function's documentation
- Choose an appropriate HTML element to generate with your function
- Write any JQuery parameters that you want use to modify your JQuery function

F.1 Identify a JQuery widget

There are many JQuery based utilities that implement a wide variety of behaviors. Some common terms for these include widgets, plugins, interactions, and events. For the purposes of this documentation, we are going to call all of these widgets. There are many to choose from, some of which are listed on http://plugins.jquery.com. If you are looking ideas, http://www.jqueryui.com hosts a number of well documented widgets with demos and sample code.

To illustrate the development of a widget, we decided to implement a collapsible tree. We wanted a widget that:

- Is visually appealing
- Can represent the tree data as an XHTML structure
- Has interesting optional implementations for later development

A search for JQuery tree led us to plugins.jquery.com which listed three pages of results. We investigated a number of them and selected the jsTree widget, found at http://www.jstree.com, as the one that best met our criteria.

One additional thing to consider in the selection of a widget is its type of licensing. Most plugins include documentation regarding licensing and reuse. In jsTree's case, it is licensed under the terms of the GNU General Public License (GPL) version 2 or the MIT license. This means it is free to use, modify and redistribute.

Install Files

Each JQuery plugin requires one or more files to work. These might include:

- The core JQuery library
- Additional script libraries, like the JQueryUI plugin

- Style sheets
- Other files

The documentation on your widget should let you know what files to download and where to get them from.

MiServer has a server level folder to store your widget's files, ServerRoot/Plugins/JQuery. To make the widget available to your MiSite, its files need to be copied into SiteRoot/Scripts. If a plugin requires multiple files or subdirectories, put them in a containing folder, like SiteRoot/Scripts/jsTree.

All JQuery utilities require the core JQuery library (`jquery.js`), which is included in `Server/Plugins`. By default, jsTree requires its own script file (`jquery.jstree.js`), a style sheet and several image files. jsTree has a number of optional features that require additional files. Since we are only using the base functionality, in this example there is no need to include them.

It is generally a good idea to keep the directory structure supplied with a plugin intact. The plugin may expect files to be in certain relative locations.

Create a Utility File

To avoid potential conflicts with the Dyalog-supplied scripts, you should create your own scripted utility file to contain your code. Locating this file in ServerRoot/Utils will ensure that it is loaded when MiServer is started.

```
)ed ⍟MyWidgets
]save MyWidgets 'C:\MiServer\Utils'
```

F.2 Writing the Code for your Widget

Now it is time to build the APL function that creates the HTML and JQuery scripts for your plugin.

It is useful to have an example to build from. Many JQuery plugins have demo pages that include source code.

jsTree uses a where it builds and displays the tree. The tree data can come from a number of sources, one of which being an HTML hierarchical unordered list, an <ul> tag structure, contained within the .

The basic rules for HTML unordered lists:

- Each list item is wrapped in <li> tags
- All list items are contained with <ul> tags

The jsTree widget requires a specific format for the list, and gives the below example of the required syntax in its documentation:

```
<li>
```

```
  <a href="some_value_here">Node title</a>
    <!-- UL node only needed for children - omit if there are no children -->
  <ul>
    <!-- Children LI nodes here -->
  </ul>
</li>
```

There are nuances that may not be readily clear from this brief example. These include:

- Each list item must be wrapped with <a> tags
- Children are designated by <ul> lists contained with the <li> tags of a parent list element
- The entire list must be wrapped by <ul> tags and then placed within the jsTree

However, the source code provided additional insight. Here is an excerpt:

```
<head>
  <script type="text/javascript" src="http://static.jstree.com/v.1.0pre/jquery.js"> </script>
  <link type="text/css" href="http://static.jstree.com/v.1.0pre/_docs/syntax/style.css"/>
  <script type="text/javascript" src=" http://static.jstree.com/v.1.0pre/jquery.jstree.js"></script>
</head>

<body>
  <div id="demo1" class="demo" style="height:100px;">
    <ul>
      <li id="phtml_1">
        <a href="#">Root node 1</a>
        <ul>
          <li id="phtml_2">
            <a href="#">Child node 1</a>
          </li>
          <li id="phtml_3">
            <a href="#">Child node 2</a>
          </li>
        </ul>
      </li>
      <li id="phtml_4">
        <a href="#">Root node 2</a>
      </li>
    </ul>
  </div>
  <script type="text/javascript" class="source below">
    $(function () {
      $("#demo1").jstree({ });
    });
  </script>
</body>
```

This view of the code much more clearly demonstrates the list syntax expected by jsTree.

F.3 Building your Widget

Many JQuery widgets have a large number of options which implement various types of behavior. Trying to encapsulate a consistent interface for all of these behaviors can be a daunting task. With

this in mind, it may be sensible to pick a few options that meet your most common needs and build your cover function around them.

With this in mind, the example below does not attempt to implement the full scope of jsTree's functionality, but rather serves as an example and starting point for how this can be accomplished.

MyWidgets.jsTree

```
∇ html←{req}jsTree pars;id;levels;items;jqpars;diff;isparent;end;repeat;li;err
    ⍝ Uses the jsTree plugin to create a tree view from hierachical data.
    ⍝ For more information on the jsTree plugin see http://www.jstree.com
    ⍝
    ⍝ req - the request object (see HTTPRequest)
    ⍝ pars - id levels items jqpars
    ⍝  id - the id attribute of the tree container
    ⍝  levels - n-element vector indicating the 'depth' of each item
    ⍝  items - n-element vector with content for each item
    ⍝  jqpars - parameters for the jsTree widget

    pars←{2>|≡⍵:,⊂,⍵ ⋄ ⍵}pars
    id levels items jqpars←4↑pars,(⍴pars)↓''⍬'' '' ''
    :If 0∊⍴id ⋄ id←'tree' ⋄ :EndIf

    :If 9=⎕NC'req' ⍝ Add JQuery links if req passed as left argument
        #.JQ.IncludeJQuery req
        req.Style'/Scripts/jsTree/themes/classic/style.css'
        'src="/Scripts/jsTree/jquery.jstree.js"'req.Script''
    :EndIf

    diff←2-/levels,1↑,levels
    err←'A child item cannot be more than one level below its parent.'
    Err ⎕SIGNAL (¯1∨.>diff)/600

    isparent←0>diff ⍝ Which items are parents?
    end←0⌈diff ⍝ How many </ul></li> tags will be at the end of each list item?
    repeat←{(⍵×⍴α)⍴α} ⍝ A function to apply those tags

    li←('<li id="'∘,¨((id,'_')∘,¨⍝ ¨⍳⍴levels)),¨⊂'"><a href="#">'
    html←li,¨items,¨('</a></li>' '</a><ul>')[1+isparent]
    html←{⎕ML←1 ⋄ ∈⍵}html,¨('</ul></li>')∘repeat¨end
    html←NL,('div id="',id,'"')#.HTMLInput.Enclose'ul'#.HTMLInput.Enclose html

    html,←#.JQ.JQueryfn'jstree'id jqpars
  ∇
```

F.4 Stepping through the Code

We chose to implement the jsTree plugin in a manner consistent with the functions in the Dyalog-supplied JQ namespace.

Initialization

```
∇ html←{req}jsTree pars;id;levels;items;jqpars;diff;isparent;end;repeat;li;err
    Uses the jsTree plugin to create a tree view from hierachical data.
    For more information on the jsTree plugin see http://www.jstree.com
    
    req - the request object (see HTTPRequest)
    pars - id levels items jqpars
     id - the id attribute of the tree container
     levels - n-element vector indicating the 'depth' of each item
     items - n-element vector with content for each item
     jqpars - parameters for the jsTree widget

    pars←{2>|≡ω:,⊂,ω ◊ ω}pars  If pars is simple, enclose
    id levels items jqpars←4↑pars,(⍴pars)↓'' ⍬ '' ''
    :If 0∈⍴id ◊ id←'tree' ◊ :EndIf  If no id, id is tree
```

jsTree takes up to four parameters, which if not defined are given a default value:

- The id of the container `<div>` tag (default is 'tree')
- The depth of each item – an n-element vector of level numbers (default is '')
- The content of each item – an n-element vector of character vectors (default is '')
- The jsTree plugin parameters - since we are only using the default options for jsTree, we will not need to pass this parameter (default is '')

Associating Scripts and Style Sheets

You need to include references to the necessary scripts and style sheets in order to make the plugin work.

The JQ namespace has two functions that will include all the references for the core JQuery library as well as the JQueryUI plugin. These are:

- JQ.IncludeJQuery – inserts a reference to the core JQuery library
- JQ.IncludeJQueryUI – inserts a reference to the core JQuery library, the JQueryUI plugin and the JQueryUI CSS

Use req.Style and req.Script to insert references to any additional style sheets or scripts, respectively (see the HTTPRequest reference for details).

```
    :If 9=⎕NC'req'
        #.JQ.IncludeJQuery req  Adds script link to JQuery core
        req.Style'/Scripts/jsTree/themes/classic/style.css'
        'src="/Scripts/jsTree/jquery.jstree.js"'req.Script''
    :EndIf
```

If the request object, req, has been passed as the left argument, references to the two scripts and the style sheet the jsTree plugin requires will be included. If req is not supplied, the references are not inserted and jsTree will simply return HTML. This is useful for debugging.

Building the HTML

```
    diff←2-/levels,1↑,levels  The difference in level between items
    err←'A child item cannot be more than one level below its parent.'
```

```
    err ⍎SIGNAL (¯1∨.>diff)/600
```

Test the levels parameter. A hierarchical list cannot have an item that is more than one level deeper than its preceding item.

```
    isparent←0>diff ⍝ Which items are parents?
    end←0⌈diff ⍝ How many </ul></li> tags will be at the end of each list item?
    repeat←{(⍵×⍴⍺)⍴⍺}⍝ A function to assist in applying those
```

Identify the starts and ends of each level change.

```
    li←('<li id="'∘,¨((id,'_')∘,¨⍕¨⍳⍴levels)),¨⊂'"><a href="#">'
```

We build the opening tags for each item. Each tag has a unique id.

```
    html←li,¨items,¨('</a></li>' '</a><ul>')[1+isparent]
```

Apply the appropriate closing tags and concatenate the opening tags to each item.

```
    html←{⍎ML←1 ◊ ∈⍵}html,¨('</ul></li>')∘repeat¨end
```

Close all groups of children with tags.

```
    html←'ul'#.HTMLInput.Enclose html
    html←NL,('div id="',id,'"') #.HTMLInput.Enclose html
```

Enclose the list in a <ul> tag and a with the supplied id attribute.

Append the function call

You need to add the JQuery function call that invokes your plugin. JQ.JQueryfn builds the Javascript which calls your plugin with whatever parameters you have supplied. The first two parameters are the name of the function as recognized by the plugin's script, in this case 'jstree,' and the id that the HTML element that the script will affect. Finally, we pass a variable for additional options that are recognized by the plugin.

```
    html,←#.JQ.JQueryfn'jstree'id jqpars
```

Testing Your Plugin

To test the function's HTML output pass it all normal parameters except for the request object.

If the HTML you have produced is XHTML compliant you can use ⍎XML to produce nicely formatted output. You can validate this output against sample HTML source code supplied by the plugin provider.

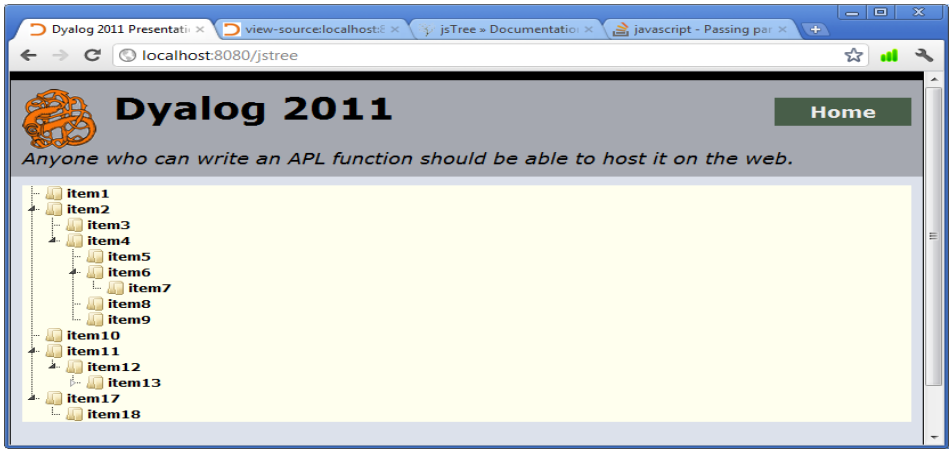
```
⍎XML ⍎XML MyWidgets.jsTree'tree'(1 2 2 1)('Item1' 'Item2' 'Item3' 'Item4')
<div id="tree">
 <ul>
  <li id="tree_1">
```

```
    <a href="#">Item1</a>
    <ul>
      <li id="tree_2">
        <a href="#">Item2</a>
      </li>
      <li id="tree_3">
        <a href="#">Item3</a>
      </li>
    </ul>
  </li>
  <li id="tree_4">
    <a href="#">Item4</a>
  </li>
 </ul>
</div>
<script type="text/javascript">/* */ $(function(){$("#tree").jstree({});});
 /* */
</script>
```

Add the Widget to a MiPage

Here we simply add the function in to our HTML. Notice that we are calling the jstTree plugin with an additional option.

```
:Class jsTreePage : MiPage

   ∇ Render req;levels;items
     :Access Public
     levels← 1 2 2 3 3 4 3 3 1 1 2 3 4 5 6 1 2
     items←'item'∘,¨⍕¨⍳⍴levels
     jqpars←'core:{animation:0}'
     req.Return req #.MyWidgets.jsTree 'tree' levels items jqpars
   ∇

:EndClass
```


Dyalog 2011 Presentati
view-source:localhost:8
jsTree » Documentatio
javascript - Passing par
localhost:8080/jstree
Dyalog 2011
Home
Anyone who can write an APL function should be able to host it on the web.
item1
item2
item3
item4
item5
item6
item7
item8
item9
item10
item11
item12
item13
item17
item18

Appendix G: Additional Features

G.1 Additional Features and the problems they solve

Features	The issues which they address
Virtual Directories	You have a directory outside the site directory from which you would like to serve resources
Resource Mapping	You have MiPages or client side script interface functions which require various and sometimes overlapping scripts and style sheets
HTTP Caching	You have multiple static resources that are called frequently by browsers, increasing load on server
HTTP Encoding /Compression	You would like to implement data encoding schemes
Identifying Content Types	You would like to identify the kind a file's type to the client browser

G.2 Extending MiServer's Reach - Virtual Directories

Normally, a client browser cannot request files outside the served directory. However, you may have good reason for wanting a directory outside of your MiSite which browsers can access.

A virtual directory is a directory which can be accessed by a browser using an alias. When the alias directory is requested, the path is interpreted on the server level as another directory. While `http://localhost:8080` might direct you to `C:\MyMiSite`, `http://localhost:8080/Plugins` might be interpreted as the directory `C:\MiServer\Plugins`.

This gives the client access to the entire directory. Access control for these directories and their subdirectories can be controlled with the SimpleAuth extension.

Virtual directories are defined in `Virtual.xml`. Below is an excerpt of the one included with the demo site.

```
<Virtual>
```

```
<!-- valid replacements are
     %ServerRoot% - MiServer root directory
     %SiteRoot%   - web site root -->
  <directory>
    <alias>JQuery</alias>
    <path>%ServerRoot%\PlugIns\JQuery\</path>
  </directory>
</Virtual>
```

To create a new virtual directory, choose an alias and a path and append them to the file using the above syntax. The path may be absolute or relative to either the `SiteRoot` or the `ServerRoot` as described above.

G.3 Simplifying Script and Style Calls - Resource Mapping

If one of your functions builds content which requires the use of a JavaScript library or a style sheet, tags which specify file paths of those resources will need to be included in the `<head>` element of your MiPage. This is easily accomplished with req.Script and req.Style. However, if the name or location of the files may change overtime or by machine, you will need to change the code.

MiServer solves this problem by keeping the path information in a configuration file called Resources.xml. The file associates a set of scripts and stylesheets with a name that can be passed to the method req.Uses. The specified resources will be appended to the head element of the page. Duplicates will be removed.

Below is an excerpt of the `Resources.xml` file found in the Demo MiSite.

Note that all paths are relative to the `SiteRoot`. In the example below you might notice something familiar from the last section. Each of the paths is using a virtual directory.

```
<Resources>

  <resource>
    <name>JQuery</name>
    <script>/JQuery/jquery-latest.js</script>
  </resource>

  <resource>
    <name>JQueryUI</name>
    <uses>JQuery</uses>
    <script>/JQuery/JQueryUI/jquery-ui.js</script>
    <style>/JQuery/JQueryUI/Themes/redmond/jquery-ui.css</style>
  </resource>

<Resources>
```

Each `resource` element must have:

- a `name` element
- any number of one or more of the following:
  - a `uses` element – the name of another resource mapping which will be also associated with the page
  - a `script` element – the file path of a Javascript library
  - a `style` element – the file path of a stylesheet

Again, to use a resource mapping, pass its name to `req.Uses`.

```
req.Uses 'JQueryUI'
```

G.4 Reducing Requests for Static Resources - HTTP Caching

When your resources are requested by a browser, it takes some time to download. If you are using static resources that will not change per download (i.e. scripts and style sheets) you might consider marking those resources to be cached by the browser. If the client's browser settings permit, this will save the files on client machine memory to be accessed locally instead of by server request.

`Server.xml` contains the setting `HTTPCacheTime`. It can be set to off, 0, or a positive integer describing the number of minutes the resource is to be cached.

G.5 HTTP Content Encoding and Compression Schemes

Content encoding is typically used to apply some lossless compression scheme to the data that's exchanged between the server and the browser.

A browser includes as part of its request for the resource an HTTP header called `Accept-Encoding`, which lists the encoding schemes that it understands and accepts. If the server supports one or more encoding schemes it can apply those encoding schemes and return the encoded data back with an HTTP header `Content-Encoding` which lists the applied encodings. The receiving browser decodes and processes the data.

MiServer currently supports deflate compression, which is supported by all major browsers.

The content encoding extension, `ContentEncoder`, has been designed to allow additional content encoding schemes to be incorporated in the future, either by Dyalog or the user community. If the user wants to implement an additional content encoding scheme, MiServer has a defined interface found in the extensions folder with which new content encoder classes can be defined. The details of such an implementation go beyond the scope of this manual, however one may look at the code in the `deflate` and `MildServer` classes to see how content encoding is implemented.

G.7 Identifying Content Types

When a browser requests a file from a server, it knows very little about that file. Servers are expected to include in the response a heading which identifies the "Media Type", formally known as MIME types, so the browser can appropriately handle the file. More information on and the official list of Media Types can be found on the Internet Assigned Numbers Authority website, here: `http://www.iana.org/assignments/media-types/index.html`

MiServer has a server level configuration file, `ServerRoot/Core/ContentTypes.xml`, which identifies the file extensions which will be associated with certain content types.

The following is an excerpt of the configuration file in the example MiSite. Note that each Media Type is contained in a `<content>` element, which itself contains two elements:

- `<ext>` - a comma separated list of one or more extensions which will be associated with the type
- `<type>` - a Media Type as defined by the official IANA list

Excerpt from ContentTypes.xml

```
<ContentTypes>
  <content>
    <ext>htm,html</ext>
    <type>text/html</type>
  </content>
  <content>
    <ext>css</ext>
    <type>text/css</type>
  </content>
  <content>
    <ext>jpeg,jpg</ext>
    <type>image/jpeg</type>
  </content>
</ContentTypes>
```

Appendix H: Server.xml Settings

H.1 General Configuration Settings

Note: All the configuration settings in Server.xml are loaded into the #.Boot.ms.Config namespace as variables.

Name
Name of the MiSite

Default: `MiServer`

This is the name of your MiSite. While it does nothing on its own, it can be used in your MiPages or MiPage template to set the title of the page.

```
req.Title #.Boot.ms.Config.Name
```

ClassName
Name of the Server Skin

Default: `DemoServer`

`ClassName` sets the name of the class found in a scripted file in `SiteRoot/Core`. This is the MiServer Skin which the site will use.

`MildServer` may be entered here to use the base server functionality

Lang
Language of the MiSite

Lang specifies your HTML Web Document language, using ISO 639 language codes. It is a website design best practice, as it helps accessibility software to properly decode your site for users with disabilities.
A few examples:

- en – English
- it – Italian
- nl – Dutch

- ru – Russian
- fr –French

Default: en

Port — The Port MiServer Listens On

This number is the port on which MiServer will listen for incoming connections. Please make sure that this port is unused by other programs on your computer, as that will cause MiServer to crash.

Default: 8080

NOTE: Port 80 is the default port number used by HTTP servers. If you don't already have a web server installed you might want to use 80 to avoid having to specify a port number when browsing the site. This may require additional permissions

SessionHandler — Identify Session Handler

Default: SimpleSessions

The name of a session handling class, found in one of the scripted files in `ServerRoot/Extensions` which contains the session handling extension.

Currently, SimpleSessions is the only session handler available for MiServer.

Authentication — Identify Authentication Handler

The name of a user authentication extension class. It must be found in one of the scripted files in `ServerRoot/Extensions`.

Default: SimpleAuth

Logger — Identify Server Logger

The name of the server logger extension class. It must be found in one of the scripted files in `ServerRoot/Extensions`.

Default: LumberJack

UseContentEncoding Toggle Content Encoding

This configuration parameter toggles the use of content encoding. A valid content encoding extension will need to be specified in the `SupportedEncodings` parameter as well.

Default: 1

Valid:

- 1 – Enable Content Encoding
- 0 – Disable Content Encoding

SupportedEncodings Encodings MiServer Supports

The name of a class containing a valid content encoder. If an encoding scheme of the same name is specified by the request in its `Accepted-Encodings` header, the scheme will be used on all outgoing content with the exception of compressed images.

Default: deflate

LogMessageLevel Sever level logging

MiServer logs a number of events and statuses. Each of these logs is a message identified by a level number. The default behavior of MiServer is to output these messages to the session. This setting allows you to throttle which messages are displayed.

Default: -1

Valid:

- -1 – all messages
- 0 – no messages
- The sum of any of the below messages have only those types of messages displayed
    - 1 – Important/Error Messages
    - 2 – Warnings
    - 4 – Informational
    - 8 – Transactional (Related to HTTP requests)
    - 16 – Compression Related Messages

Default Page Resource Called When None Specified

If a client navigates to a directory on your server but does not specify a resource, MiServer checks to see if there is a file whose name matches the text of this parameter. You must specify a default page. This name will be the name looked for in each directory.

Default: index.dyalog

HTTPCacheTime Sever level logging

Sets the time in minutes in which static resources (HTML pages, script libraries, style sheets, etc.) cued to be cached on the client side. If the browser has caching turned off, this will have no effect.

Default: 0

Valid:

- 0 – Off
- A number of whole minutes

IdleTimeOut Time Until Idle Behavior

You may wish to have behaviors that trigger when the server has been idle for a period of time. A value in `IdleTimeOut` sets the amount of time the server is idle before it executes the method MildServer.Idle.

Default: 0

Valid:

- 0 – Off
- A number of whole minutes

H.2 Error Trapping / Debugging Configuration Parameters

The following parameters are associated with DrA, MiServer's error trapping utility. Any DrA behavior requires `TrapErrors` to be set to 1.

TrapErrors
Trap Server Errors

This setting determines whether errors which occur during runtime are trapped and logged by the DrA utility or cause the server to crash. This setting must be turned on for any of the following settings to be pertinent.

If set to 1, DrA will generate error logs in component files, storing them in the `/DrA` directory. Additionally, it can be set to email those logs.

Default: 0

Valid:

- 0 – Crash
- 1 – Trap and log errors with the DrA utility

Debug Set
Debugging Behavior

Debug allows the data collected by DrA to be displayed in the browser, providing diagnostic information for the developer. Debug can also be set to allowing live editing of the resource through the browser.

If a MiPage generates an untrapped error, a diagnostic page is returned that can detail the error and allows the page to be edited from the browser window.

Note: TrapErrors must be 1 for the following behaviors.

Default: 0

Valid:

- 0 – No Debug Info
- 1 – Debug Info – The DrA utility generates a diagnostic page which includes information about the MiPage
- 2 – Allow Editing – The diagnostic page includes a link which opens the editpage

MailMethod Set Debugging Behavior

DrA can email log messages. The email can be sent in a number of ways, including via SMTP server, the .Net interface and Outlook.

Default: NONE

Valid:

- NONE – Do not send the error messages/logs via email
- SMTP – Uses an SMTP server to send mail, requires a valid value for the SMTPGateway parameter
- NET – Uses .Net to send the email
- OUTLOOK – Sends the email with Microsoft Outlook

Note: The NET and OUTLOOK values both require the .Net framework to be installed

MailRecipient Set Debugging Behavior

The email address to which DRA will send the log.

Default: No Default Value

SMTPGateway SMTP Address

The SMTP gateway which DrA will use to email log messages, if `MailMethod` is set to SMTP.

Default: No Default Value

Appendix I: The Future of MiServer

I.1 The MiServer Project

The MiServer Project is an open source project to promote the development of MiServer and serve as a community building exercise.

We want people to use, talk about, modify, experiment with, and extend MiServer. When they do, would like them to share their extensions, widgets and any other modifications with the community at large. The MiServer page at APLWiki at `http://www.APLWiki.com/MiServer` will be a repository for community contributed content, as well as where we will distribute the 'official' release of MiServer. We will also publish MiServer in the Dyalog Library.

We are excited to see what we can build together.