Overview

Event handling allows you to specify what actions are to be taken when a particular event occurs in a MiPage. Examples of events include:

- When the value of an input field is changed
- When a button is clicked
- When the mouse hovers over a particular page element.

There are two fundamental ways to handle events in a web application –

- In the browser – this generally involves writing some JavaScript for your event handler
- Communicate the event back to the server – unlike a native GUI application where all elements of the user interface are directly accessible from the application code, the separation between the browser and the server can require some thought about what information to send to the server from the client.

MiServer event handling allows you to use either method. MiServer's event handling is based upon jQuery's on() method. For more advanced event handling, it's a good idea to read up on jQuery on().

Simple Example

The following code implements a simple MiPage that contains a button and a <div> that displays the number of times the button is clicked.

```
:Class HandlerSample : MiPage

    Clicked←0   ⍝ track the number of time the button is clicked

    ∇ Compose;btn
      :Access public
      btn←Add _.button'Click Me' ⍝ add a button to the page
      btn.On'click'              ⍝ define a 'click' handler on the button
      '#output'Add _.div Clicked ⍝ add a <div> to contain the response from the server
    ∇

    ∇ r←APLJax
      :Access public
      Clicked+←1                 ⍝ increment the number of times clicked
      r←'#output'Replace Clicked ⍝ and refresh the content of the <div>
    ∇

:EndClass
```

This illustrates the basic steps you will generally use to create an event handler:

1) Create some HTML content – in this case a button -
   `btn←Add _.button'Click Me'`

2) Create a handler that's bound to the HTML content -
   `btn.On'click'`
   In creating the handler, you'll generally specify:
   - What events handler will react to ('click' in this case)
   - Whether the event will be handled in the client or on the server – by default events are handled on the server by a method named `APLJax`
   - If the event is to be handled on the server, what data is to be sent to the server. In this example we don't need to send any data to the server.

   There are other event handler attributes that you can specify, but we'll cover those later on.

3) Write the callback code for the handler.

```
∇ r←APLJax
  :Access public
  Clicked+←1                 ⍝ increment the number of times clicked
  r←'#output'Replace Clicked ⍝ and refresh the content of the <div>
∇
```

   If the event is handled on the server, the callback code can send a response back to the client to do things like update the web page's content or execute code in the web page. In this case we replace the contents of the element with id="output" (the div we created in Compose) with the new number of clicks.

Specifying an Event Handler

There are two techniques to specify an event handler:

- Use the “On” method for an element.

```
btn←'b1' Add _.button 'Click Me'
btn.On 'click'
```

- Add a Handler to the object

```
Add _.Handler '#b1' 'click'
```

`_.Handler`

All pages, HTML elements, and widgets in MiServer are based on the **`HtmlElement`** class. The **`HtmlElement`** class has a public field, `Handlers`, which is a vector of the event handlers bound to the element. The event handlers are instances of the **`_.Handler`** class.

`_.Handler` *Public Fields*

The table below summarizes the public fields in **`_.Handler`**. Fields you're more likely to use are listed higher in the table. Fields listed in "grayed" rows are for more advanced use and not likely to be used in most cases. More detailed discussion follows.

Field	Description	Default Value
`Selector`	Either: • A character vector of the jQuery/CSS selector(s) for the elements • A 2 element vector of character vectors of [1] the jQuery/CSS selector(s) for the elements [2] the jQuery/CSS selector(s) for delegated elements • A reference to the element to which the event handler is to be bound	`''`
`Events`	The event(s) for which handler will respond	`''`
`Callback`	One of: • 1 – handle the event on the server by calling the APLJAX function • `'name'` – handle the event on the server by calling the user defined function `name` • 0 – do not callback to the server, handle the event in the client	1
`ClientData`	Specifies the data you want sent to the server from the client	`''`
`Delegates`	A character vector of the jQuery/CSS selector(s) for delegated elements (can also be specified as the second element of `Selector`)	
`JavaScript`	JavaScript to execute. If making a callback to the server this is executed prior to making the callback.	`''`
`Page`	The URL to which the server callback is directed.	current page

`Hourglass`	Boolean which indicates whether to turn the cursor to an hourglass while the server executes the handler.	1
`jQueryWrap`	Boolean which indicates whether to wrap the handler definition in code such that it will be defined and bound when the web page is loaded.	1
`ScriptWrap`	Boolean which indicates whether to wrap the handler definition in a <script> element.	1
`ForceInternal`	Indicates whether to treat the event as an internal event in a widget. One of: • ¯1 – determine by seeing if the event is an element of the widget's InternalEvents list • 0 – do not treat it as an internal event • 1 – force it to be treated as an internal event	¯1
`WidgetDef`	Used to define the syntax for specific types of information for different JavaScript utility libraries.	jQuery

`Selector`

`Selector` specifies what elements the event listener is to be bound to. For instance, assume we've created the following div element.

```
myDiv←'#myid' '.myclass' Add _.div 'Click Me'
```

When using **`myDiv.On`** to specify the handler, **`Selector`** is set for you.

```
myHandler ← myDiv.On 'click'
```

If you specify Selector yourself it can be:

- A reference to the element
  ```
  myHandler ← Add _.Handler myDiv 'click'
  ```
- A jQuery/CSS selector (see jQuery Selectors)
  - Bind the listener to the element with id="myid"
    ```
    myHandler ← Add _.Handler '#myid' 'click'
    ```
  - Bind the listener to all elements with class="myclass"
    ```
    myHandler ← Add _.Handler '.myclass' 'click'
    ```
  - Bind the listener to all <div> HTML elements
    ```
    myHandler ← Add _.Handler 'div' 'click'
    ```
  - You can specify multiple selectors
    ```
    myHandler ← Add _.Handler '#myid, td' 'click'
    ```
    Will bind the listener to the element with id="myid" AND all <td> elements.
- An empty vector `''` or `'⍎document'` – to bind the handler to the entire document.
  ```
  myHandler ← Add _.Handler '' 'click'
  ```

Events

Events specifies what events the event listener should listen for. It is a space-delimited list of events.

```
    myHandler.Events←'click'              ⍝ listen for click
    myHandler.Events←'click dblclick'     ⍝ listen for click and dblclick
```

The events that you can specify include jQuery and HTML events. When using HTML events, you can drop the leading "on" – in other words, instead of "oncopy" you would specify "copy" for the event name.

You can also define your own events and use jQuery's trigger() method to trigger them in the client – see _.Timer for an example of this.

Callback

Callback specifies where the event is to be handled. Valid values for Callback are:

- 1 (the default) – handle the event on the server by calling a function named APLJax
- 'Name' – handle the event on the server by calling a function named Name
- 0 – handle the event in the client

If you have a simple page with a single handler, using the default behavior and using APLJax. However, if you have several handlers you may want to write separate

ClientData

ClientData specifies what information is to be sent from the client to the server.
See the section on ClientData found later in this document.

Delegates

Delegates are used in jQuery's event handling framework primarily to address two circumstances.

- When the same event is to be bound to a large number of elements. For instance, every cell in a large table. Rather than internally bind a handler to every cell, it's more efficient to bind one handler to the table element itself and then specify the cells as delegates.
- When dynamic elements need handlers bound to them. Event handlers are bound when the page is loaded. If, after it's loaded, a page creates new elements that need event handling, then you must use delegates.

See https://learn.jquery.com/events/event-delegation/ for a discussion on delegates.

Hourglass

Hourglass is a Boolean which indicates whether to change the cursor to an hourglass when an event is handled by the server. This is intended for event whose server processing time may be lengthy. Most events will be handled so quickly that flipping between the normal cursor/hourglass and back again isn't noticeable. The default value is 1.

`Page`

`Page` specifies the URL to which to direct the event handling. As in the example at the beginning of this document, most events will be handled in the same page that rendered the original web content. However, if you want to separate your event handling code to another page, set `Page` to the URL of

`JavaScript`

`JavaScript` is any JavaScript code you would like to be run in the client **before** the event is signaled to the server.

`jQueryWrap`

`jQueryWrap` is a Boolean which specifies whether to wrap the handler in jQuery syntax so that the handler is defined and bound when the web page loads. The default is 1.

This setting is for advanced use when you want to generate and manipulate the handler's code for separate use.

`ScriptWrap`

`ScriptWrap` is a Boolean which specifies whether to wrap the handler in an HTML <script> element. The default is 1.

This setting is for advanced use and is generally used when you want to combine several pieces of JavaScript into a single <script> element.

`ForceInternal`

`ForceInternal` indicates whether the event should be handled as a widget's "internal event". This setting applies only to 3rd party widgets (jQuery and Syncfusion).

Widgets in JavaScript libraries like jQueryUI and Syncfusion will often have a list of events – we call these "internal events" because the handler for them is defined within the widget's parameters. Internal events are handled slightly differently and may have different arguments. The MiServer APIs for these JavaScript libraries have a public field, `InternalEvents`, which lists the events that are internal for the widget at the time the MiServer API was written.

The default setting for `ForceInternal` is `¯1` which indicates to check `InternalEvents` for whether to treat the event as an internal event.

A setting of `1` is used to the force the event to be treated as an internal event. You would set this in the (unlikely) case when the widget has an event that is not listed in InternalEvents. This might happen if you install an updated version of a JavaScript library and new events have been introduced.

A setting of `0` means do not treat the event as an internal event even if it's listed in `InternalEvents`. It's unlikely that you'd ever use this setting, but it's included for completeness.

In general, the nuances of internal events are not important to the MiServer user. But, it is recommended that the user review the source documentation for a widget to be familiar with which events are internal and what features may be of interest.

`WidgetDef`

`WidgetDef` specifies the syntaxes used by a particular widget library for

- the internal event handler syntax
- the syntax to access the event object
- the syntax to access the widget's object model
- the syntax to access the widget itself

WidgetDef is primarily of interest to people who may want to incorporate additional JavaScript widget libraries into MiServer.

_.Handler Constructor

```
h←Add _.Handler [Selectors [Events [Callback [ClientData [Delegates
              [JavaScript [Page]]]]]]]
```

All arguments to the constructor are positional and optional.

Event Handling – `ClientData`

`ClientData` allows you to specify data that is to be sent back to the server. The data can be associated with any element on the page, not just the element to which the event handler is bound.

By default the callback mechanism will return:

By default the callback mechanism will return: `_event`	the name of the event
`_what`	the id/name, if one exists, of the element that triggered the event
`_value`	the value, if one exists, of the element that triggered the event
`_selector`	the selector specified for the handler
`_target`[1]	the id, if one exists, of the innermost element that the event fired upon. This will normally be the same as `_what`.
`_currentTarget`[1]	the id, if one exists, of the element to which the handler was bound

These fields are directly available in your page when you handle an event.

If you do not specify **`ClientData`**, MiServer will also automatically serailize and return any form data found on the page. If you wish to send other information and any form data, you will need to use "serialize" as described below.

[1] For an example on the use of `_target` and `_currentTarget`, see the /Examples/Techniques/TargetsExample page in the MS3 MiSite.

You can specify other information to be sent to the server using the following syntax:

```
name type [argument] [selector]
```

If the type does not use an argument, then selector can be in the 3rd position.
In other words:

`'data' 'val' '' '#myinput'` and
`'data' 'val' '#myinput'` are equivalent **because val does not use an argument**.

`name`	the name to give the data on the server side
`selector`	jQuery/CSS selector of or a reference to the element from which to get the data If `selector` is omitted, use the element to which the handler is bound if applicable
`type`	the type of data to return

`type`	**Returns**
`attr`[2]	an HTML element attribute
`prop`[1]	a JavaScript DOM property
`css`	a CSS setting
`is`	specific settings – see jQuery.is()
`html`	the HTML content of the element
`val`	the value of the element (generally applies only to form elements)
`option`	for jQuery and Syncfusion widgets, this will retreive the value of the option specified in `argument`
`method`	for jQuery and Syncfusion widgets, this will return the result of executing the widget method named in **`argument`**
`event`	for jQuery widgets and HTML elements return a stringified representation of the event object for Syncfusion widgets, return a stringified representation of the argument passed to the event handler
`argument`	same as **`event`**
`this`	return a stringified representation of the element which triggered the event
`model`	for jQuery and Syncfusion widgets, this will return either the stringified representation of the entire widget model, or element specified in **`argument`**
`ui`	same as **`model`**
`eval`	the result of the evaluation of a JavaScript string specified in **`argument`** by using the JavaScript `eval()` function.
`js`	the result of executing the JavaScript specified in **`argument`** This differs from **`eval`** in that the JavaScript is executed inline rather than using the JavaScript `eval()` function.
`string`	a constant string
`serialize`	the serialization of all input elements in all forms on the page, or the form specified in **`selector`**
`⍎`	the result of executing the `1↓` of **`type`**. This is a shorthand version of **`js`**

[2] It's useful to understand the difference between HTML attributed and DOM properties.- for a good discussion on this, see http://lucybain.com/blog/2014/attribute-vs-property/

`argument`	dependent on **`type`**

`type =`	`argument =`	Example
`attr`	the attribute to return	`'attr' 'title'`
`css`	the CSS setting to return	`'css' 'font'`
`html`	"	`'html'`
`is`	the setting to return – see jQuery.is()	`'is' ':checked'`
`val`	"	`'val'`
`eval`	the JavaScript string to evaluate	`'eval' '2+2'`
`string`	the string to return	`'string' 'constant'`
`event`	the element of the event object	see jQueryUI document
`ui`	the element of the ui object	see jQueryUI document
`model`	the element of the model object	see Syncfusion document
`argument`	the element of the argument object	see Syncfusion document
`serialize`	"	`'serialize'`
`js`	the JavaScript to execute	`'js' 'alert("Hello");'`
`⍎`	"	`'⍎alert("Hello");'`

Example:

```
h←Add Handler              ⍝ add an event handler
h.ClientData←('content' 'html' '#div1')
              ('bgcolor' 'css' 'background-color' '#div2')
```

Returns

- a variable named "content" which contains the HTML content of the element with id "div1"
- a variable named "bgcolor" with the CSS background color of the element with id "div2"

Event Handling - Retrieving Client Data from Callback Functions

Client data can be retrieved in two basic ways:

- Define a public field in your MiPage with the same name as the data you want to retrieve
  For instance, in the specification used above
  ```
  h.ClientData←('content' '#div1' 'html')
                ('color' '#div2' 'css' 'background-color')
  ```
  You could defined two fields and they would be populated automatically.
  ```
  :field public content
  :field public color
  ```

- The other way is to use the **`Get`** method to retrieve the data by name. The left argument to **`Get`** is the default value to use if the data element is not found. Again, using the example from above, the following could be used.
  ```
  '???' Get 'content'
  'white' Get 'color'
  ```

Event Handling – Sending Responses Back to the Client

There are four functions which specify actions to be taken on the client side in response to a callback function.

```
r ← selector Replace new
r ← selector Append  new
r ← selector Prepend new
r ←          Execute javascript
```

Replace	Replaces the HTML content of the element specified by `selector` with `new`
Append	Appends `new` to the HTML content of the element specified by `selector`
Prepend	Prepends `new` to the HTML content of the element specified by `selector`
Execute	Executes javascript string (using JavaScript's eval() function)
selector	The selector of the elements to update
new	The new content with which to update
javascript	A character vector of the JavaScript to execute in the client

Example:

```
r ← '#result' Replace _html.h2('Hi')
r,← Execute 'alert("Happy Birthday!")'
```

Callback functions must return a result, though the result could be '' if no action is to be taken on the client side.