Article
Vitaliy Serdtsev · Nov 24, 2017 11m read

$(REST - CSP - (-HyperEvents) + EasyUI + File Upload). Part 2

Last time, we stopped on static stuff. Let ' s continue...

As you can see, the URL for our static things looks like this:

```
/rest/some/random/stuff/here
```

You may be tempted to add all the files that you may potentially need to the URL map, but this is wrong, so let ' s not do this.

Let's recall that " Url " in the URL map is **regular expression**, which means that you can be more flexible in programming the path for our static handler.

Something like this

```
/:some random stuff
```

won ' t work, since the parameter cannot contain the "/" character, which we may actually want to use multiple times. Therefore, all we can use is this

```
/(.*)
```

which denotes any character found 0 and more times.

Note:
More information about regular expressions can be found here: Regular Expressions.

Let ' s modify our class:

```
XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <Route Url="/(.*)" Method="GET" Call="StaticFiles"/>
  <Route Url="/" Method="GET" Call="MainPage"/>
</Routes>
}

ClassMethod StaticFiles(url) As %Status
{
  // not doing anything useful yet
  q $$$OK
}
```

Let ' s now refresh the page and oops!

Our button has disappeared and there *seem* to be no errors...

That ' s all because the very first URL in our URL map meets the requirements of all possible paths: not just current ones, but future ones as well, so everything will be processed by the **StaticFiles** method.

This is why there is a corresponding warning in the documentation:

Documentation:
Caché compares the incoming REST URL with the Route URL property or the Map Prefix property. It starts at the first Route or Map element and continues to test each following element until it finds a match. Once it finds a match it either sends the incoming call to the call specified in the Route or forwards the URL to the class specified in the Map. In both cases, it ignores any elements in the Routes after the matching element; consequently, the order of the elements in the Routes is significant. If an incoming URL could match multiple elements of the Routes, Caché uses the first matching element and ignores any subsequent possible matches.

Let ' s fix our class by swapping the methods in the URL map.

Handling of static data

If we take a look at the headers of requests for static data – not just for our still unfinished application, but for other applications as well – we will see that they contain tons of parameters: path, type, date stamp of the cached file (HTTPIFMODIFIEDSINCE) and much more.

On the server side, we need to use the passed URL to determine the name of the file being requested, its type, size, encoding and date of modification.

Based on this, we will then need to generate corresponding response headers and place the file itself into the response body.

As you can see, you don ' t need to do much. Let ' s start...

No!!! Let me first write it this way:

```
ClassMethod StaticFiles(url) As %Status
{

// let's delegate this work to %CSP.StreamServer, although we could do it ourselves
  k %request.Data
  s %request.Data("FILE",1)=%request.URL

  d ##class(%CSP.StreamServer).OnPreHTTP()
  d ##class(%CSP.StreamServer).OnPage()
  q $$$OK
}
```

If you feel that standard processing with the %CSP.StreamServer class is not enough – for instance, you need additional checks for file access rights – you can do it using **%CSP.StreamServer** sources as an example. All right, let ' s make sure that statics load without errors now.

Now we have hyperevents on the agenda.

Hyperevents

Hyperevents were a part of Caché when the very term Ajax wasn ' t particularly popular and common, which was a really long time ago.

Let ' s take a look at hyperevents in the framework of the Caché DBMS (key aspects):

1. They allow you to easily and transparently call server (Caché) methods from the client side (typically from JavaScript);
2. They can be called synchronously [*#server*] or asynchronously [*#call*];
3. The format of sent/received data is a string, which means that more complex types (such as JSON) need to be converted to a string and the reconverted on the server;
4. They can return JS code for execution on the client side;
5. They can hide the name of the server method that we want to run, so that no one can substitute their own code and run it.

Let ' s now examine how it all works.

First, let ' s change the handler for our button:

```
<button onclick="#server(..Add({?:'1 pear',?:'5 apples'}))#">Click me</button>
```

and take a look at the generated HTML code:

```
<button onclick=
"cspHttpServerMethod('pXZzvVt0xEkSN4SAJoQbFJIfTLt_wD$J2ghItig6wlo-',{?:'1 pear',?:'5
apples'})">Click me</button>
```

As you can see, the method call is replaced with a call to the client function **cspHttpServerMethod**, which accepts our parameters, and the name of the method called is encrypted with a session key that never leaves the Caché server.

The *#call* call is identical to the *#server* call, except for the name of the client function: in this case, it will be **cspCallHttpServerMethod**.

These two functions are defined in the *cspxmlhttp.js* file, which, along with *cspbroker.js*, is used for developing a web application with CSP/CLS.

On the server side, hyperevents are handled by the **%CSP.Broker** class, which decrypts the full name of the method and calls it with the passed parameters.

So how can we improve hyperevents in relation to our REST application?

To begin with, let ' s define two client functions **cspHttpServerMethod** and **cspCallHttpServerMethod** – using AJAX.

To do this, let ' s use the jQuery library that we have already added to our application.

Let ' s add the following code to the class:

```
ClassMethod WriteCommonJS() [ Internal, Private ]
{
  &html<<script type="text/javascript">

// our improved hyperevents. We can't do anything with their names, since they are ge
nerated automatically by #server/#call commands
function cspHttpServerMethod(method,params)
{
  $.ajax({
    url: method,
    data: params,
    async: false,
    success: function(data){alert($.toJSON(data,null,2))}
  });
```

```
}

function cspCallHttpServerMethod(method,params)
{
  $.ajax({
    url: method,
    data: params
  });
}

(function($){

  $.ajaxSetup({
    type: 'POST',
    dataType: 'json',
    cache: false,
    async: true,
    contentType: 'application/json; charset=UTF-8',
    processData: false,
    beforeSend: function(jqXHR,settings) {
      settings.data=$.toJSON(settings.data);
// prior to sending, let's convert a JSON objects into a JSON string
    },
    statusCode: {
      // try playing with access methods to see these errors
      401: function() {alert('You are not authorized')},
      404: function() {alert('Page not found')},
      405: function() {alert('Forbidden method')},
      500: function(jqXHR,textStatus,errorThrown) {alert(errorThrown)}
    },
    error: function(jqXHR, textStatus, errorThrown){
      if (textStatus==='timeout') {alert('Ran out of time :(')}
    }
  });

})(jQuery);
</script>>
}
```

Note:
I moved the code to a separate method to avoid cramming the MainPage method.

From now on, whenever the **csp(Call)HttpServerMethod** is called, any method passed as the first parameter will be called using AJAX. The name of the method will be already encrypted.

We ' ll be passing data back and forth in the JSON format with the help of the POST method in the request body.

One method will be called synchronously, the other one asynchronously.

Note:
The full list of available parameters and the methods of the AJAX function can be found in the official jQuery documentation: jQuery.ajax. YouTube also features lots of jQuery and JavaScript lessons.

Now you need to call our new **WriteCommonJS** method. The simplest way to do it is to use the one-line #[]# command:

```
...
```

```
  <script type="text/javascript" src=
"easyui/js/jquery.fileupload.js"></script>#[d ..WriteCommonJS()]#
...
```

Let ' s not forget the **Add** method:

```
ClassMethod Add(ByRef args As %ZEN.proxyObject) As %Status
{
  s r=args.? + args.?
  d args.%Clear()
  s args.Result=r
  q $$$OK
}
```

Here we get our JSON object by reference (in Caché, this is going to be a **%ZEN.proxyObject** class object) and use it to return the result into some field.

Now, if you click the button, we will get a "405 / Forbidden method" error. This happens because we haven ' t added a handler for our hyperevents to the URL map, which causes the static handler to be called.

However, it waits for the GET method, and we, as you may remember, are calling hyperevents with POST, hence the error.

Let ' s update the URL map:

```
XData UrlMap [ XMLNamespace = "http://www.intersystems.com/urlmap" ]
{
<Routes>
  <Route Url="/:hyperevent" Method="POST" Call="RestHyperEvent"/>
  <Route Url="/" Method="GET" Call="MainPage"/>
  <Route Url="/(.*)" Method="GET" Call="StaticFiles"/>
</Routes>
}
```

Our full URL for calling a hyperevent will look like this:

```
http://localhost:57772/rest/pXZzvVt0xEkSN4SAJoQbFJIfTLt_wD$J2ghItig6wlo-
```

The last part will be different for different sessions and/or methods.

Note:
Since the name of the method is encrypted and encoded, it cannot contain "/" characters, that ' s why this method of path definition works just fine.

Let ' s also add the handler itself:

```
ClassMethod RestHyperEvent(hyperevent) As %Status [ Final, Internal ]
{
  #dim %request As %CSP.Request
  #dim params As %ZEN.proxyObject
  #dim ex As %Exception.AbstractException

  q:hyperevent="" $$$ERROR($$$CSPBadBrokerRequest)
  q:%session.NewSession $$$ERROR($$$CSPSessionTimeout)
```

```
  try{
    s hyperevent=$lg($zcvt(..Decrypt(hyperevent),"I","UTF8"))
  }catch{
    return $$$ERR($$$InvalidDecrypt)
  }
  q:hyperevent="" $$$ERR($$$CSPIllegalRequest)
  s find=$find(hyperevent,":")
  s:find hyperevent=$e(hyperevent,1,find-2)
  s cls=$p(hyperevent,".",1,$l(hyperevent,".")-1),method=$e(hyperevent,$l(cls)+2,*)
  q:cls="" $$$ERR($$$ClassNameRequired)
  q:method="" $$$ERROR($$$MethodNameRequired)
  s cls=$$$NormalizeClassname(cls)

  d ##class(%ZEN.Auxiliary.jsonProvider).%ConvertJSONToObject(%request.Content,,.
params)

  s st=$$$OK
  try{
    $$$ThrowOnError($classmethod(cls,method,.params))
    d params.%ToJSON(,"aelo")
  }catch(ex){
    if ex.Name="<METHOD DOES NOT EXIST>" {
      s st=$$$ERROR($$$MethodDoesNotExist,method)
    }elseif ex.Name="<CLASS DOES NOT EXIST>" {
      s st=$$$ERROR($$$ClassDoesNotExist,cls)
    }else{
      throw
    }
  }
  q st
}
```

This is a service method and should not be replaced by you, unless you act very carefully.

Here we decrypt the name of the passed parameter, run various checks, convert the JSON string into an object and call the method itself; once done, we write the result in the JSON format to the response body.

Let ' s click our button again and see that the result is 0, although it should be 6.

If we add some debugging information to the beginning of method **Add**:

```
d args.%CopyToArray(.arr)
k ^tmp
m ^tmp("arr")=arr
```

we ' ll see some gibberish:

```
USER>zw ^tmp
^tmp("arr","Ð"_$c(144))="1 pear"
^tmp("arr","Ð"_$c(146))="5 apples"
```

This happens because the request body is not yet automatically converted into the necessary encoding.

Important:
The upcoming version of the Caché DBMS should have a new parameter controlling this behavior.
Update: already fixed, see **CONVERTINPUTSTREAM**.

Let ' s fix it. The best option is to do it in the **OnPreDispatch** callback method that is called in parallel with each request, so avoid overloading it with too many " heavy " operations:

```
ClassMethod OnPreDispatch(
  pUrl As %String,
  pMethod As %String,
  ByRef pContinue As %Boolean) As %Status
{

  #dim %request As %CSP.Request

  If $IsObject(%request.Content),$zcvt(%request.Content.ContentType,"L")[
"charset=utf-8" {
    Set contentTmp=%request.Content.%ConstructClone(-1)
    Set contentTmp.Attributes("CharEncoding")="UTF8"

    Merge attr=%request.Content.Attributes
    Do %request.Content.Clear()
    Merge %request.Content.Attributes=attr

    Do %request.Content.CopyFrom(##class(%IO.MetaCharacterStream).%New(contentTmp))
  }

  Quit $$$OK
}
```

Everything looks good now:

```
USER>zw ^tmp
^tmp("arr","?")="1 pear"
^tmp("arr","?")="5 apples"
```

and we get 6 on the page.

The next part will be about integrating jQuery File Upload, but first...

#REST API #UI Development #Frontend #Caché

Source URL:https://community.intersystems.com/post/rest-csp-hyperevents-easyui-file-upload-part-2