Artículo
Ricardo Paiva · 17 jul, 2020 Lectura de 13 min

Programación especial con InterSystems

¡Hola desarrolladores!

Apuesto a que no todos los que estáis familiarizados con InterSystems Caché conocéis las extensiones de Studio para trabajar con el código fuente. De hecho, podéis usar Studio para crear vuestro propio tipo de código fuente, compilarlo en código de objeto e interpretable (INT) y a veces hasta añadir soporte para finalización de código. Es decir, teóricamente podéis hacer que Studio acepte cualquier lenguaje de programación a ejecutar por el DBMS (Sistema de Administración de Bases de Datos), así como Caché ObjectScript. En este artículo, os mostraré un ejemplo sencillo de cómo escribir programas en Caché Studio usando un lenguaje similar a JavaScript. Si os interesa, seguid leyendo.

Si vais al namespace SAMPLES, encontraréis un ejemplo de trabajo con tipos de archivo definidos por el usuario. El ejemplo sugiere abrir un documento del tipo " Example User Document (.tst) " type, y solo hay un archivo de este tipo llamado TestRoutine.TST, que, de hecho, se genera sobre la marcha. La clase requerida para trabajar con este tipo de archivo se llama *Studio.ExampleDocument*. No vamos a entrar en detalle en este ejemplo y vamos a crear uno propio. El tipo de archivo ".JS" ya está siendo usado en Studio, y el JavaScript que queremos soportar no es exactamente el JavaScript original. Vamos a llamarle CacheJavaScript y el tipo de archivo será ".CJS". Para empezar, cread una clase *%CJS.StudioRoutines* como subclase de la clase *%Studio.AbstractDocument* y añadidle el soporte del nuevo archivo.

```
/// El nombre de la extensión, puede ser una lista de extensiones separadas por coma
si esta clase acepta más de una
Projection RegisterExtension As %Projection.StudioDocument(DocumentDescription = "Cac
héJavaScript Routine", DocumentExtension = "cjs", DocumentIcon = 1, DocumentType = "J
S");
```

- DocumentDescription — se muestra como la descripción del tipo en la ventana de abrir archivo en la lista de filtros;
- DocumentExtension — la extensión de los archivos que serán procesados por esta clase;
- DocumentIcon — el número de icono empieza en cero; los siguientes iconos están disponibles:
- DocumentType — este tipo se usará para resaltar código y errores; están disponibles los siguientes tipos:
  - INT — código INT de Cache Object Script
  - MAC — código MAC de Cache Object Script
  - INC — incluir macro en Cache Object Script
  - CSP — página de servidor Cache (Cache Server Page)
  - CSR — regla de servidor Cache (Cache Server Rule)
  - JS — código JavaScript
  - CSS — hoja de estilo HTML
  - XML — documento XML
  - XSL — transformación XML
  - XSD — esquema XML
  - MVB — código mvb básico multivalor
  - MVI — código mvi básico multivalor

Ahora implementaremos todos los métodos necesarios para soportar el nuevo tipo de código fuente en Studio.

Los métodos **`ListExecute`** y **`ListFetch`** se usan para obtener una lista de archivos disponibles en el namespace y para mostrarlos en el diálogo de abrir archivo.

```
ClassMethod ListExecute(ByRef qHandle As %Binary, Directory As %String, Flat As %Bool
ean, System As %Boolean) As %Status
{
    Set qHandle=$listbuild(Directory,Flat,System,"")
    Quit $$$OK
}

ClassMethod ListFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd As %I
nteger = 0) As %Status [ PlaceAfter = ListExecute ]
{
    Set Row="",AtEnd=0
    If qHandle="" Set AtEnd=1 Quit $$$OK
    If $list(qHandle)'=""||($list(qHandle,4)=1) Set AtEnd=1 Quit $$$OK
    set AtEnd=1
    Set rtnName=$listget(qHandle,5)
    For {
        Set rtnName=$order(^rCJS(rtnName))    Quit:rtnName=""
        continue:$get(^rCJS(rtnName,«LANG»))'=«CJS»
        set timeStamp=$zdatetime($get(^rCJS(rtnName,0)),3)
        set size=+$get(^rCJS(rtnName,0,«SIZE»))
        Set Row=$listbuild(rtnName_".cjs",timeStamp,size,"")
        set AtEnd=0
        set $list(qHandle,5)=rtnName
        Quit
    }
    Quit $$$OK
}
```

Guardaremos la descripción de los programas en el global **^rCJS**, y el método **ListFetch** recorrerá este global para devolver cadenas que tengan lo siguiente: nombre, fecha y tamaño del archivo encontrado. Para que los resultados se muestren en el cuadro de diálogo, deberéis crear un método **Exists** que compruebe si existe un archivo con ese nombre.

```
/// Devolver 1 si existe la rutina 'name' y 0 si no existe.
ClassMethod Exists(name As %String) As %Boolean
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Set rtnNameExt = $piece(name,".",$length(name,"."))
    Quit $data(^rCJS(rtnName))&&($get(^rCJS(rtnName,«LANG»))=$zconvert(rtnNameExt,«U»
))
}
```

El **TimeStamp** devolverá la fecha y hora del programa. El resultado también se muestra en el cuadro de diálogo de abrir archivo.

```
/// Devolver el registro de la hora de la rutina 'name' en formato %TimeStamp. Esto s
e usa para determinar si la rutina ha
/// sido actualizada en el servidor y por tanto debe volver a cargarse desde Studio.
Entonces el formato debería ser  $zdatetime($horolog,3),
/// o "" si la rutina no existe.
ClassMethod TimeStamp(name As %String) As %TimeStamp
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Set timeStamp=$zdatetime($get(^rCJS(rtnName,0)),3)
    Quit timeStamp
```

```
}
```

Ahora tendremos que cargar el programa y guardar los cambios en el archivo. El texto del programs, línea a línea, se guarda en el mismo global **^rCJS**.

```
/// Cargar la rutina en Name hacia el flujo Code
Method Load() As %Status
{
    set source=..Code
    do source.Clear()
    set pCodeGN=$name(^rCJS(..ShortName,0))
    for pLine=1:1:$get(@pCodeGN@(0),0) {
        do source.WriteLine(@pCodeGN@(pLine))
    }
    do source.Rewind()
    Quit $$$OK
}

/// Guardar la rutina almacenada en Code
Method Save() As %Status
{
    set pCodeGN=$name(^rCJS(..ShortName,0))
    kill @pCodeGN
    set @pCodeGN=$ztimestamp
    Set ..Code.LineTerminator=$char(13,10)
    set source=..Code
    do source.Rewind()
    WHILE '(source.AtEnd) {
        set pCodeLine=source.ReadLine()
        set @pCodeGN@($increment(@pCodeGN@(0)))=pCodeLine
    }
    set @pCodeGN@(«SIZE»)=..Code.Size
    Quit $$$OK
}
```

Ahora viene la parte más interesante: compilar nuestro programa. Lo compilaremos en código INT y así tenemos una compatibilidad total con Caché. Este artículo es tan solo un ejemplo, y por eso usé solo una pequeña fracción de las funcionalidades de CachéJavaScript: declaración de variables (`var`), lectura (`read`), y salida de datos (`println`).

```
/// CompileDocument se llama cuando se quiere compilar el documento
/// En este punto, ya ha llamado a los hooks del control de fuente
Method CompileDocument(ByRef qstruct As %String) As %Status
{
    Write !,«Compile: „,..Name
    Set compiledCode=##class(%Routine).%OpenId(..ShortName_“.INT»)
    Set compiledCode.Generated=1
    do compiledCode.Clear()

    do compiledCode.WriteLine(" ;generated at "_$zdatetime($ztimestamp,3))
    do ..GenerateIntCode(compiledCode)

    do compiledCode.%Save()
    do compiledCode.Compile()
    Quit $$$OK
}
```

```
Method GenerateIntCode(aCode) [ Internal ]
{
    set varMatcher=##class(%Regex.Matcher).%New("[ \t]*(var[ \t]+)?(\w[\w\d]*)[ \t]*(
\=[ \t]*(.*))?")
    set printlnMatcher=##class(%Regex.Matcher).%New("[ \t]*(?:console\.log|println)\(
([^\)]+)\)?")
    set readMatcher=##class(%Regex.Matcher).%New("[ \t]*read\((.*)\,(.*)\)")

    set source=..Code
    do source.Rewind()
    while 'source.AtEnd {
        set tLine=source.ReadLine()

        set pos=1
        while $locate(tLine,"(([^\'\""\;\r\n]|[\'\""][^\'\""]*[\'\""])+)",pos,pos,tCo
de) {
            set tPos=1
            if $zstrip(tCode,"*W")="" {
                do aCode.WriteLine(tCode)
                continue
            }
            if varMatcher.Match(tCode) {
                set varName=varMatcher.Group(2)
                if varMatcher.Group(1)'="" {
                    do aCode.WriteLine($char(9)_«new „_varName)
                }
                if varMatcher.Group(3)'=“» {
                    set expr=varMatcher.Group(4)
                    set expr=..Expression(expr)
                    do:expr'="" aCode.WriteLine($char(9)_«set „_varName_“ = „_expr)
                }
                continue

            } elseif printlnMatcher.Match(tCode) {
                set expr=printlnMatcher.Group(1)
                set expr=..Expression(expr)
                do:expr'=“» aCode.WriteLine($char(9)_«Write „_expr_“,!»)

            } elseif readMatcher.Match(tCode) {
                set expr=readMatcher.Group(1)
                set expr=..Expression(expr)
                set var=readMatcher.Group(2)
                do:expr'="" aCode.WriteLine($char(9)_«read „_expr_“,»_var_",!")
            }
        }
    }
}

ClassMethod Expression(tExpr) As %String
{
    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)\+[ \t]*(?:\""([^\"
"]*)\""|\'([^\']*)\')([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1_""$2$3""$4"

    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)(?:\""([^\""]*)\""|
\'([^\']*)\')[ \t]*\+([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1""$2$3""_$4"

    set matchers($increment(matchers),«matcher»)="(?sm)([^\'\""]*)(?:\""([^\""]*)\""|
```

```
\'([^\']*)\')([^\'\""]*)"
    set matchers(matchers,«replacement»)="$1""$2$3""$4"

    set tResult=tExpr
    for i=1:1:matchers {
        set matcher=##class(%Regex.Matcher).%New(matchers(i,«matcher»))
        set replacement=$get(matchers(i,«replacement»))

        set matcher.Text=tResult

        set tResult=matcher.ReplaceAll(replacement)
    }

    quit tResult
}
```

Podéis ver el código INT generado para cada clase o programa compilado. Para ello, deberéis escribir un método GetOther. Es bastante simple: el objetivo es devolver una lista delimitada por comas de programas generados para el código fuente.

```
/// Devolver otros tipos de documentos con los que esto está relacionado.
/// Pasó un nombre y devolvéis una lista separada por comas de los otros documentos c
on los que está relacionado
/// o "" si no está relacionado con nada. Tened en cuenta que a esto se le puede pasa
r un documento de otro tipo
/// por ejemplo si vuestro documento 'test.XXX' crea una rutina 'test.INT', entonces
también será llamado
/// con 'test.INT' para que podáis devolver 'test.XXX' para completar el ciclo.
ClassMethod GetOther(Name As %String) As %String
{
    Set rtnName = $piece(Name,".",1,$length(Name,".")-1)_".INT"
    Quit:##class(%Routine).%ExistsId(rtnName) rtnName
    Quit ""
}
```

Implementamos un método para bloquear un programa de forma que solo un desarrollador a la vez pueda editar un programa o clase en el servidor.

No os olvidéis de escribir un método para eliminar programas.

```
/// Eliminar la rutina 'name', que incluye la extensión de rutina
ClassMethod Delete(name As %String) As %Status
{
    Set rtnName = $piece(name,".",1,$length(name,".")-1)
    Kill ^rCJS(rtnName)
    Quit $$$OK
}

/// Bloquear la rutina actual, el método predeterminado solo desbloquea el global ^rC
JS con el nombre de la rutina.
/// Si eso falla, devolver entonces un código de estado del error. En caso contrario,
 devolver $$$OK
Method Lock(flags As %String) As %Status
{
    Lock +^rCJS(..Name):0 Else Quit $$$ERROR($$$CanNotLockRoutine,..Name)
    Quit $$$OK
}
```

```
/// Desbloquear la rutina actual, el método predeterminado solo desbloquea el global
^rCJS con el nombre de la rutina
Method Unlock(flags As %String) As %Status
{
    Lock -^rCJS(..Name)
    Quit $$$OK
}
```

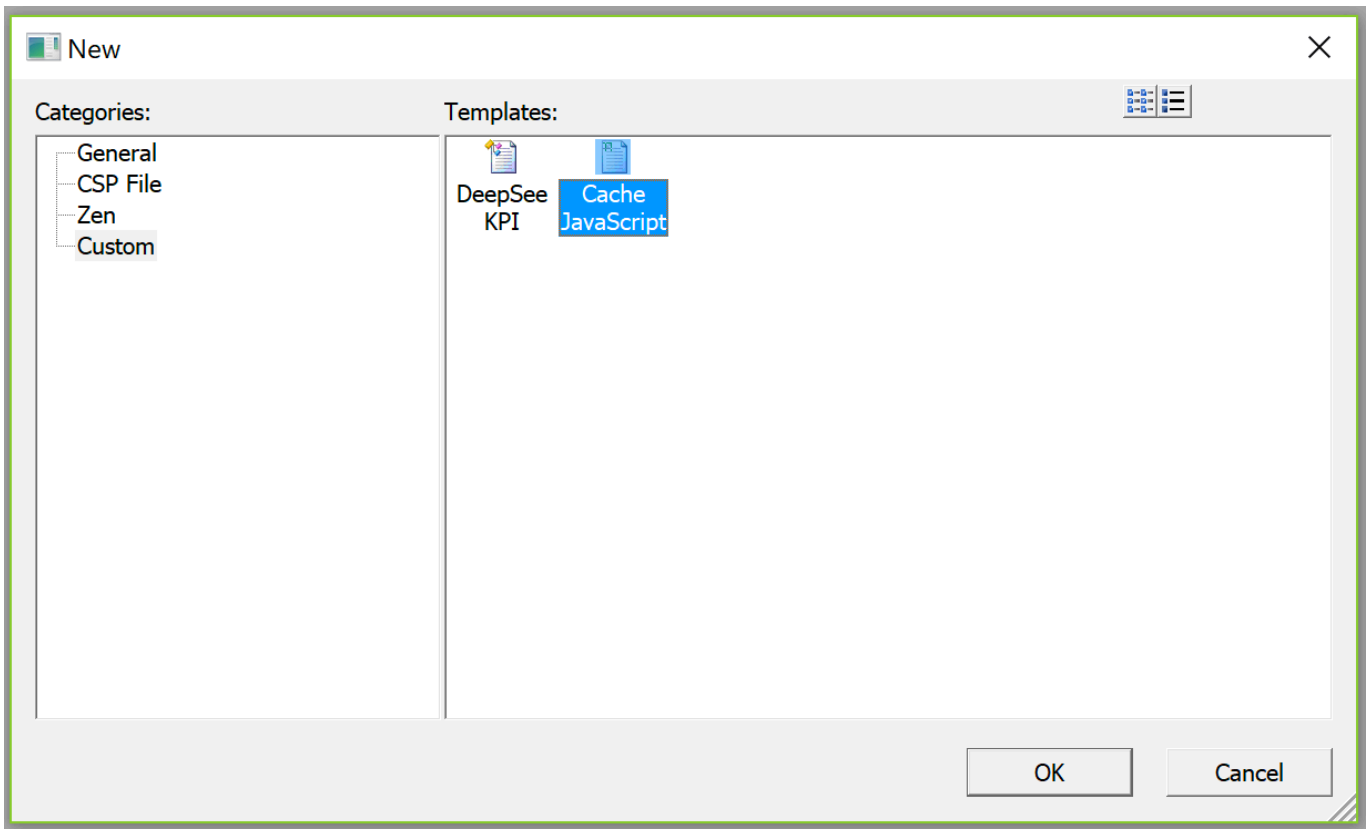
Muy bien, hemos escrito una clase que nos permite trabajar con nuestro tipo de programas. Sin embargo, no podemos escribir dicho programa aún. Vamos a arreglarlo. Studio permite definir plantillas y hay 3 formas de hacerlo: un simple archivo CSP de un formato particular, una clase CSP heredada de la clase **%CSP.StudioTemplateSuper** y, finalmente, una página ZEN heredada de **%ZEN.Template.studioTemplate**. En nuestro caso, usaremos la última opción por simplicidad. Las plantillas pueden ser de 3 tipos también: para crear nuevos objetos, solo plantillas de código y complementos, que no generan salidas.

En nuestro caso, necesitaremos una plantilla para crear nuevos objetos. Creemos una nueva clase llamada **%CJS.RoutineWizard**. Su contenido es bastante simple: hay que describir un campo para introducir el nombre del programa y luego describir el nombre del nuevo programa y su contenido obligatorio para Studio en el método **%OnTemplateAction**.

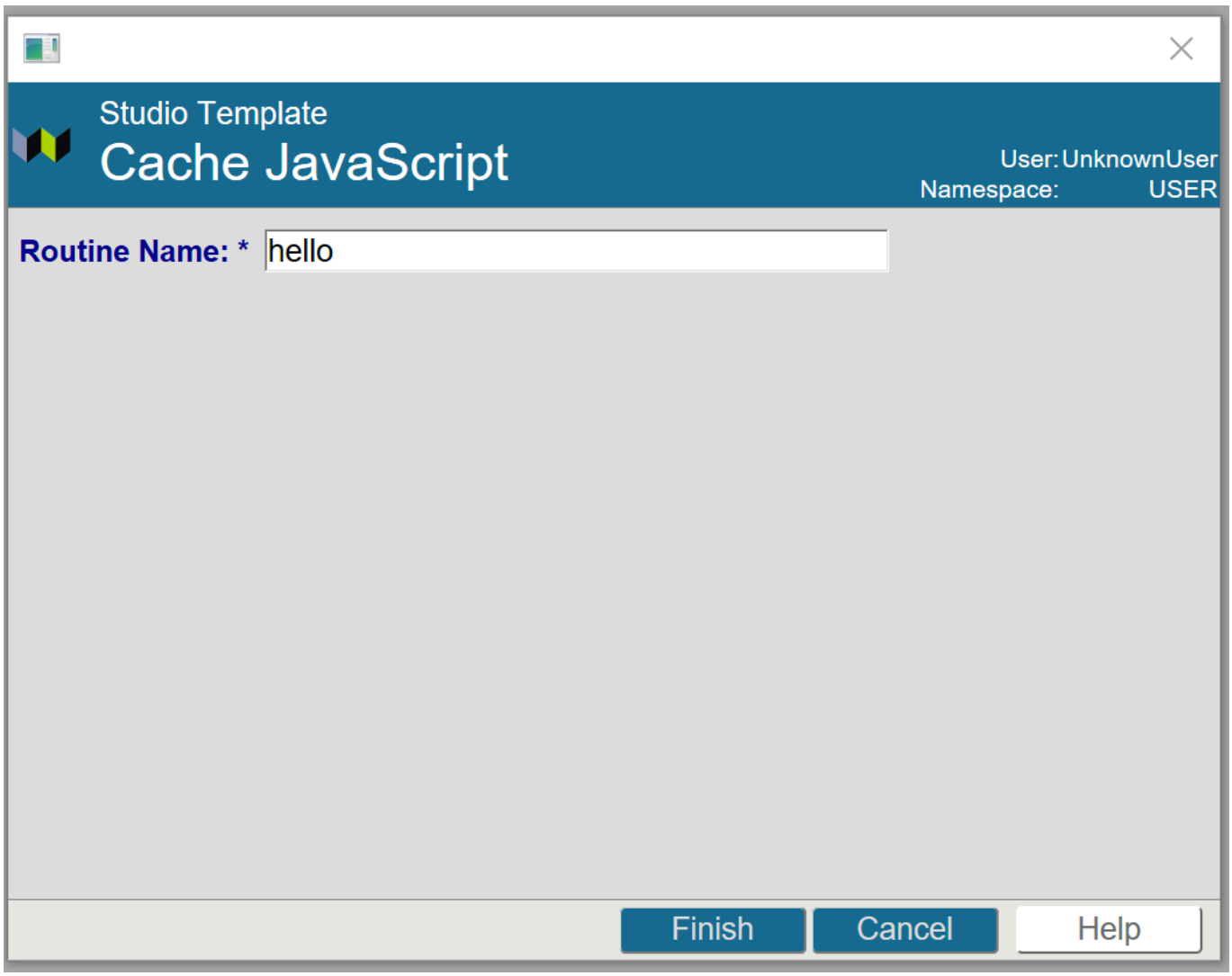
```
/// Plantilla de Studio:

/// Crear una nueva rutina Cache JavaScript.
Class %CJS.RoutineWizard Extends %ZEN.Template.studioTemplate [ StorageStrategy = ""
]
{

Parameter TEMPLATENAME = "Cache JavaScript";

Parameter TEMPLATETITLE = "Cache JavaScript";

Parameter TEMPLATEDESCRIPTION = "Create a new Cache JavaScript routine.";

Parameter TEMPLATETYPE = "CJS";

/// Qué tipo de plantilla.
Parameter TEMPLATEMODE = "new";

/// Si en este caso TEMPLATEMODE="new" entonces este es el nombre de la pestaña
/// de Studio en la que se muestra esta plantilla. Si no se especifica ninguna, enton
ces
/// se muestra en la pestaña 'Custom'.
Parameter TEMPLATEGROUP As STRING;

/// Este bloque XML block define los contenidos del panel de contenido de esta planti
lla de Studio.
XData templateBody [ XMLNamespace = "http://www.intersystems.com/zen" ]
{




}

/// Proveer contenidos del componente de descripción.
```

```
Method %GetDescHTML(pSeed As %String) As %Status
{
    Quit $$$OK
}

/// Esto se llama cuando la plantilla se muestra por primera vez;
/// Esto ofrece una oportunidad para definir el foco, etc.
ClientMethod onstartHandler() [ Language = javascript ]
{
    // asignar foco al nombre
    var ctrl = zenPage.getComponentById('ctrlRoutineName');
    if (ctrl) {
        ctrl.focus();
        ctrl.select();
    }
}

/// Controlador de validación para formulario incorporado en la plantilla.
ClientMethod formValidationHandler() [ Language = javascript ]
{
    var rtnName = zenPage.getComponentById('ctrlRoutineName').getValue();

    if ('' == rtnName) {
        return false;
    }

    return true;
}

/// Este método es llamado cuando la plantilla está completa. Cualquier
/// salida al dispositivo principal es devuelta a Studio.
Method %OnTemplateAction() As %Status
{
    Set tRoutineName = ..%GetValueByName("RoutineName")

    Set %session.Data("Template","NAME") = tRoutineName_".CJS"
    Write "// "_tRoutineName,!
    Quit $$$OK
}

}
```

Y esto es todo. Ahora ya podéis crear vuestro primer programa escrito en Caché JavaScript en Studio.



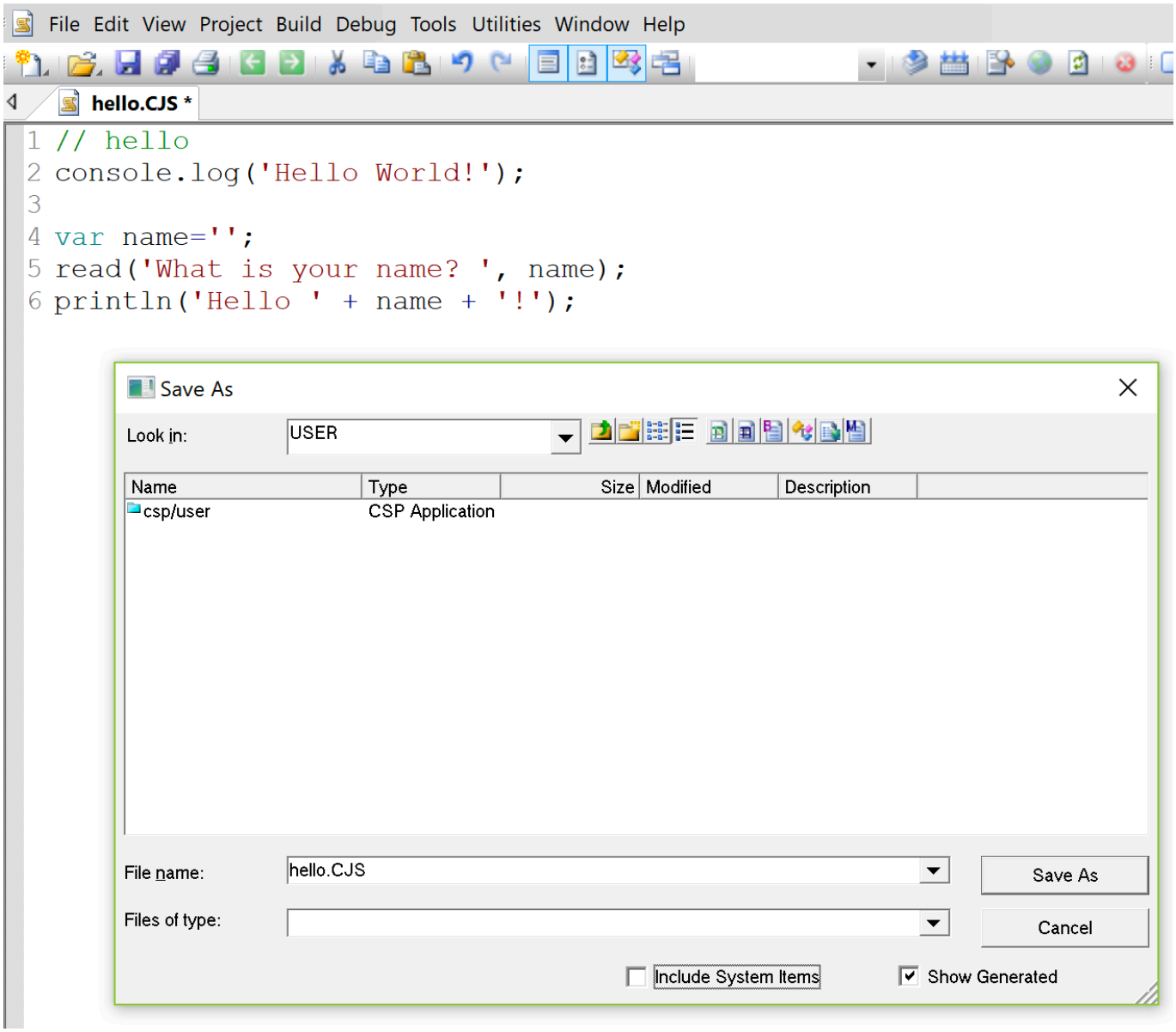
Vamos a llamarlo “ hello ” .



El código fuente en CachéJavaScript puede verse así, por ejemplo:

```
// hello
console.log('Hello World!');

var name='';
read('What is your name? ', name);
println('Hello ' + name + '!');
```

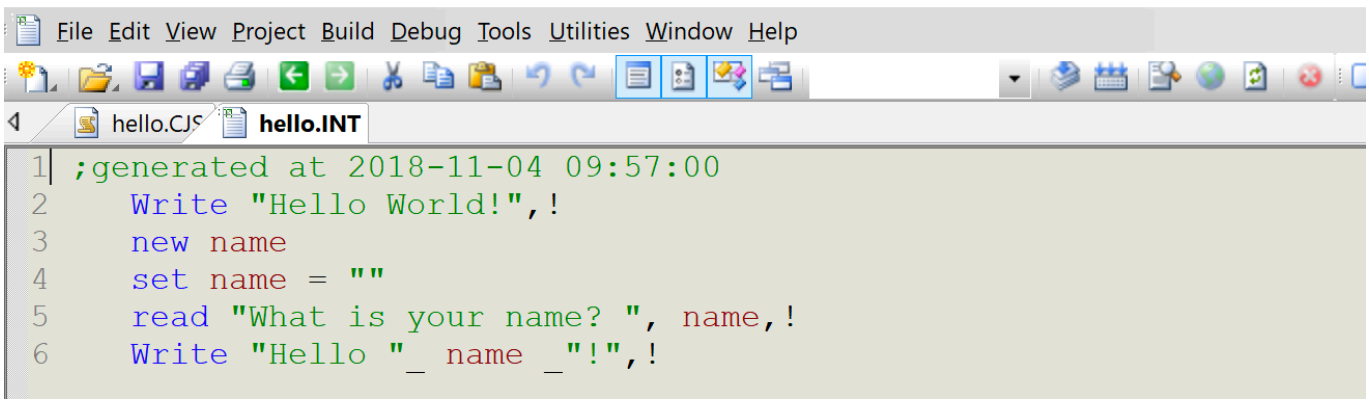
Lo guardamos.



Después de guardar y compilar veremos que el código int también se generó y compiló con éxito, en la salida:

```
Compilation started on 11/04/2018 12:57:00 with qualifiers 'ck-u'
Compile: hello.CJS
Compiling routine : hello.int
Compilation finished successfully in 0.034s.
```

Veamos otra fuente.



Ahora podemos ejecutarlo en el terminal

```
USER>d ^hello
Hello World!
What is your name? daimor
Hello daimor!
```

Así es como podéis describir cualquier lenguaje (hasta cierto punto, por supuesto) que os guste y usarlo para codificar la lógica empresarial del lado del servidor para la plataforma de datos Caché/IRIS. Si este lenguaje no es compatible con Studio, habrá problemas con el resaltado de código. Este ejemplo demuestra el funcionamiento con programas, pero puede crear clases de Caché de la misma forma. Las posibilidades son casi infinitas: solo se necesita escribir un analizador léxico y un compilador completo, y luego diseñar el mapeo correcto entre todas las funciones del sistema Caché y las construcciones específicas del nuevo lenguaje. Este tipo de programas también pueden exportarse e importarse con la compilación, como se hace con muchos otros programas en Caché.

Quien quiera hacerlo en casa, puede descargar los códigos fuente aquí, en udl o xml.

#Studio #Caché