Websockets SSL

Contenido

Introducción 2
Definición de los webSockets 2
Definición básica 2
Scopes → Ambito de actuación 3
Definiendo el Server WebSockets 4
Maneras de trabajar con los WebSockets 6
UT API 6
UT Free → Programación standard 8
Proceso de Mensajes - Backend 9
Seguridad 10
Funciones y Métodos para manejo de WebSockets 11
Certificados 12
Codigo ejemplo 13
Ejemplo de configuración servidor sin SSL 13
Ejemplo de configuración con SSL 14
Ejemplo FrontEnd – View 14
Ejemplo Backend – API 15
Resumen 17

Introducción

UT ya dispone de la capacidad de manejo de Websockets con SSL, lo que nos dará plena seguridad y confianza en nuestros programas.

Con websockets podemos recrear varios ejemplos de trabajo: notificaciones en tiempo real, estado de procesos, actualizaciones, chats,... Harbour nos brinda ahora la posibilidad de usarlos pero también hemos de ser consciente de sus limitaciones ya que no fue diseñado para este proceso.

De la misma manera que comentamos del uso de uhttpd2 como servidor web, en el cual tiene sus limitaciones pero que a la vez puede ser capaz de ofrecernos su potencia tranquilamente de manera sobrante, podemos también aplicarlo a los websockets.

Programar con websockets no es tarea fácil y he intentado seguir aplicando la filosofía UT en esta parte del proceso e intentar hacer fácil lo difícil.

Definición de los webSockets

No todas las pantallas que creemos necesitaran el uso de WS, solo que aquellas que por su diseño y objetivo tengan de tener esta funcionalidad que nos ayudará mantener la conexión activa. Así pues, aquellas que necesitemos el uso de websockets, necesitaremos definirlo al igual que definimos un control. Usaremos el siguiente comando estilo xBase, con todas sus opciones que iremos comentando a lo largo del manual.

```
DEFINE WEBSOCKET
      [ ONOPEN <cFuncOnOpen> ]
      [ ONCLOSE <cFuncOnClose> ]
      [ ONMESSAGE <cFuncOnMessage> ]
      [ TOKEN <cToken> ]
      [ SCOPE <cScope> ]
      [ LOG, TRACE ]
      OF <o>
```

Definición básica

```
DEFINE WEBSOCKET OF o
```

¡Con este simple comando ya armamos nuestra módulo!

Independiente de la definición podemos definir 3 eventos que podemos usar si necesitamos y usaremos los comandos:

```
ONOPEN <cFunction>
ONCLOSE <cFunction>
ONMESSAGE <cFunction>
```

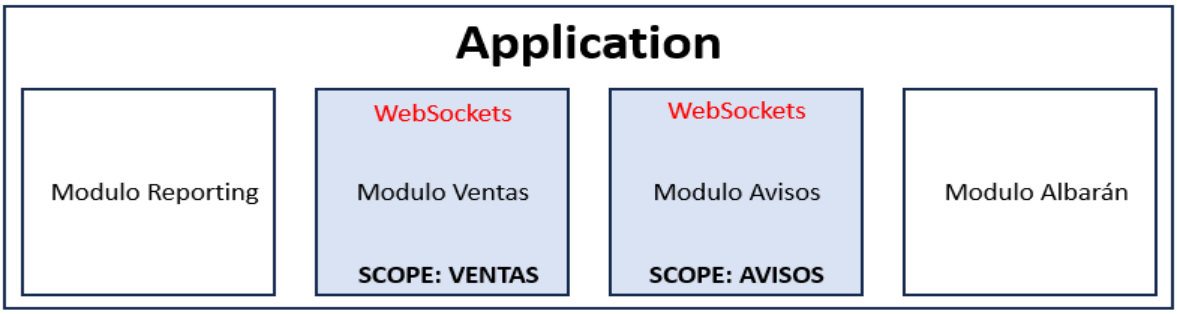
Donde <cFunction> será una función Javascript que se ejecutará cuando se dispare el evento. onOpen y onClose nos puede ir bien para conocer el estatus de la conexión. onMessage se ejecutará cada vez que el cliente reciba un mensaje. Si queremos procesar nuestro sistema nativamente lo habremos de gestionar definiendo estas funciones.

El comando LOG/TRACE activa salidas con la consola del inspector.

```
DEFINE WEBSOCKET TRACE OF o
```

Scopes → Ambito de actuación

Cuando creamos un módulo puede ser que necesitemos estar de alguna manera conectados con ciertos usuarios que usen ese módulo, similar a un chat, en el que cuando estás conectado, tu mensaje llega a todos o a un grupo. Se ha creado una cláusula cuando definimos el websocket que llamaremos SCOPE . Imaginemos que uno de los módulos que creamos es para estar en contacto directo con otros usuarios que estén usando este módulo



Este mismo esquema nos indica que si por ejemplo estoy en el módulo avisos, quiero estar notificado en cualquier momento y sólo quien esté usando este módulo comparte esta funcionalidad.

Para ello definiremos el uso websocket con el comando SCOPE

```
DEFINE WEBSOCKET SCOPE 'VENTAS'
```

Esto nos ayudará a enviar notificaciones, datos, … a todos los que en ese momento estén en ese módulo desde nuestro backend.

```
oDom:SendSocket( ……, 'VENTAS' )
```

Definiendo el Server WebSockets

El servidor WebSocket lo definiremos igual que lo hacemos con el Servidor UT. Basicamente creamos un hilo en el que definiremos nuestro servidor, propiedades y lo pondremos en marcha.

En el ejemplo proporcionado se muestra esta definición, no hay complicaciones

```
#define VK_ESCAPE 27
#define IS_SSL       .T.

function main()

      hb_threadStart( @WebServer() )
      hb_threadStart( @WebServerSocket() )

      while inkey(0) != VK_ESCAPE
      end

retu nil
```

De la misma manera que iniciamos el servidor Http con un hilo, haremos lo mismo con el servidor WS

Básicamente definimos el server con la función UWebSocket() y luego le podemos dar alguna configuración

```
function WebServerSocket()

      local oServer     := UWebSocket()

      if IS_SSL

            oServer:SetPort( 8443 )
            oServer:SetCertificate( 'cert/privatekey.pem',
'cert/certificate.pem' )
            oServer:SetSSL( .T. )

      else
            oServer:SetPort( 9000 )
      endif

      oServer:SetTrace( .T. )

      oServer:bValidate := {|hParam| MyValidate(hParam) }
      oServer:bMessage  := {|cSocket, cReq, hParam| MyDispatcher( cSocket,
cReq, hParam ) }

      // ----------------------------------------------------//

      IF ! oServer:Run()
```

```
            ? "=> WebServerSockets error:", oServer:cError

            RETU 1
      ENDIF

RETU 0
```

Métodos / Datas	Descripción
SetPort(<nPort>)	Si compilamos bajo SSL el mas usado es el 8443, sino podemos usar el 9000
SetCertificate(<cPrivateKey>, <cCertificate>)	Certificados. Por defecto usará: ● cert/privatekey.pem ● cert/certificate.pem
SetSSL(<lOnOff>)	Si usamos SSL . Default .F.
::bValidate	Codeblock que apuntará a una función para validar el token. Recibe un hash con 2 claves: ● 'token' → Token que podemos validar (opcional) ● 'scope' → Ambito del módulo (opcional) Este codeblock es opcional. El hecho de especificarlo, provoca que el servidor validará cada conexion
::bMessage	Codeblock que apuntará a una función a la que le pasará el mensaje que se envia desde el cliente. La función recibira 3 parámetros: ● 'cSocket' → string que identifica al token del cliente ● 'cReq' → string que identifica el mensaje del cliente ● 'hParam' → hash igual que bValidate

Aquí resaltar que el uso del mismo puerto para https y wss puede dar errores, pero fácilmente podemos especificar el 443 para https y el 8443 para websockets.

Maneras de trabajar con los WebSockets

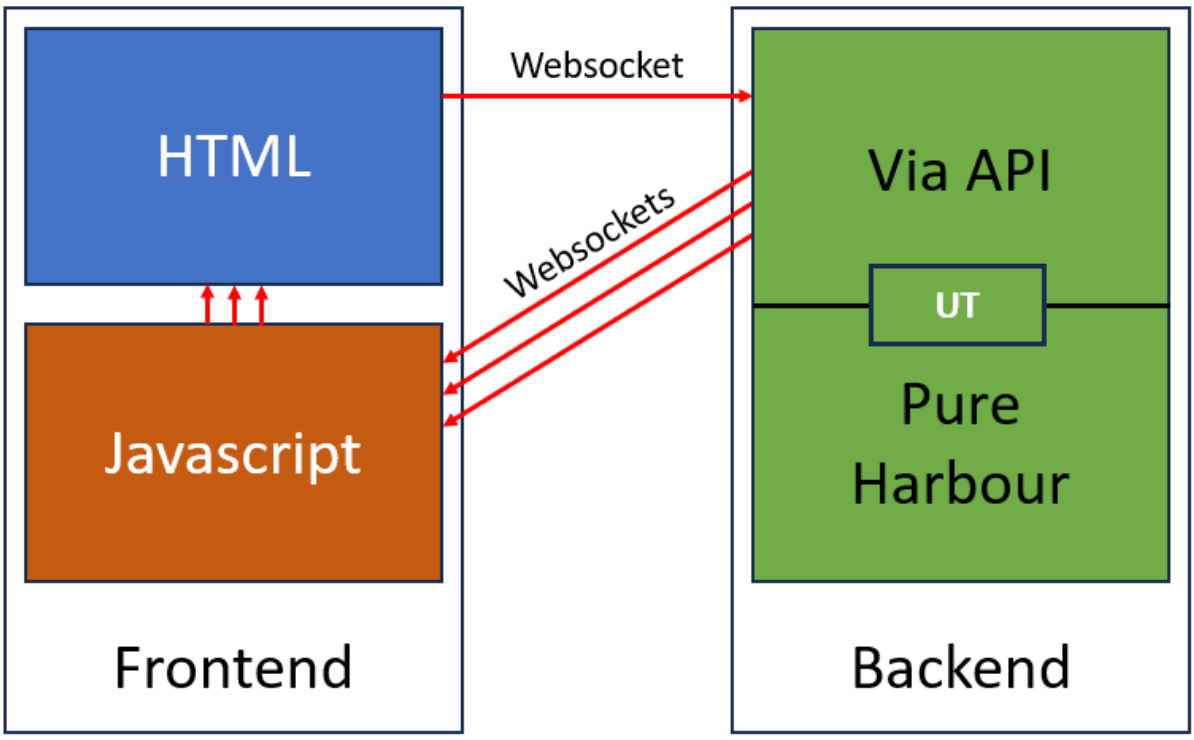
Dividiría 2 maneras de trabajar con UT y websockets.

- La clásica que llamaremos *UT API* → 1 framework tipo TWeb para diseño de pantallas y 1 Api para nuestro backend, que procese las peticiones y gracias al objeto oDom, poder dar respuesta y manejar los controles de la pantalla. Así de fácil

- La más purista que llamaremos *UT FREE* → que es usar directamente javascript para manejar los websockets y no usar la api de UT sino harbour puro y duro y manejar los diferentes mensajes de los WS, manejarlos y dar respuesta.

UT API

La primera opción podría tener una visión como la siguiente



Es la que hasta ahora venimos usando en gran medida, porque nos facilita enormemente el trabajo.

Aquí el concepto es que el control de la pantalla dispare un evento, por ejemplo, el button y esto nos conecta con nuestra api. Dentro del api ejecutamos nuestra función y con nuestro objeto oDom ejecutamos nuestra respuesta para el cliente. Ahora el objeto oDom tendrá un nuevo método para ejecutar los mensajes de los websockets.

La gran diferencia es que estos mensajes no se ejecutarán al final de nuestra función, sino que serán mensajes asíncronos que a medida que vayamos ejecutando se enviaran al momento al cliente.

Hasta ahora nosotros podemos tener en nuestro api, código parecido a este

```
function MyFunc( oDom )
      …
      oDom:SetTableData(...)
      oDom:Set( 'name', 1234 )
      oDom:Set( 'address', 'Hill Street, 14')
      …
return nil
```

Este código se ejecutará al salir de la función, es decir, no lo va ejecutando a medida que le indicamos a oDom un método.

Por lo contrario, imaginemos un caso que se nos dará muy a menudo y es el de lanzar un proceso de larga duración y queremos que nuestra pantalla se vaya actualizando e ir indicando en qué punto se encuentra para no dar una sensación de aplicación congelada (**UI freezing).**

Ahora fácilmente podemos definir por ejemplo un control de texto que nos indicará cómo va el proceso y un button para iniciar el proceso. Cuando le demos al button llamaremos al proceso "myprocess" del api "myapi".

En el api, podemos diseñar ahora el proceso tan fácil, como:

```
function MyProcess( oDom )

      // Inicio Proceso y notifico

      oDom:SendSocket( 'Iniciando…')

      …

      // Al cabo de unos segundos, sigo con el proceso y notifico

      oDom:SendSocket( 'Continuando con el proceso…' )

      …

      // Y una última ejecución

      oDom:SendSocket( 'Finalizando…' )

      …

retu nil
```

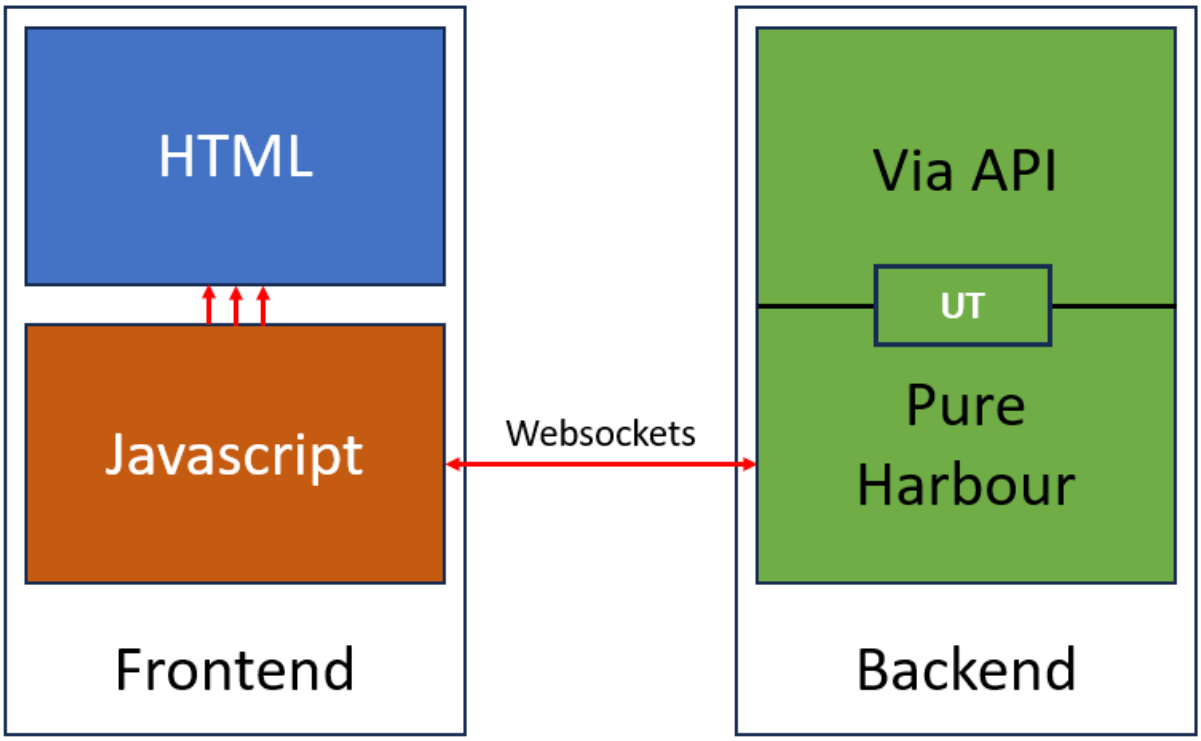
Los 3 SendSockets irán enviando mensajes a medida que se ejecuten las líneas de nuestro código. Este caso por ejemplo es muy bueno para escenarios de proceso de larga duración en el que si quieres puedes ir notificando del estado en cada momento al cliente.

Para nosotros nos resultará mágico, porque sigue con nuestra manera de trabajar consiguiendo realmente un resultado espectacular

UT Free → Programación standard

La segunda opción es la que da más juego, pero necesitas programar las diferentes maneras en que se pueden gestionar los websockets y a la vez la sencillez va cambiando a más complejidad, pero que se soluciona entendiendo todo el proceso y poniendo orden. Si de alguna manera vía api, era tan fácil como enviar el mensaje, aquí ya tenemos que crear un manejo de ellos.

Podríamos tener la siguiente visión de esta opción



Aquí trabajaremos directamente desde javascript partiendo ya desde la conexión que tengamos definida en el módulo. Conceptualmente es desde js que se envían mensajes a nuestro servidor WS.

Proceso de Mensajes - Backend

Las funciones que podremos usar para enviar sockets desde nuestro backend serán:

- SendSocket()
- SendSocketJS()

La diferencia entre las 2 funciones, es que la primera enviará un mensaje que entrará siempre en el cliente por la función que hayamos definido en la cláusula ONMESSAGE <cFunction> a la hora de definir el WebSocket. La segunda podemos elegir a qué función de javascript enviaremos el mensaje, lo que nos puede dar mucho juego a la hora de crear nuestra funcionalidad.

Cuando definimos nuestro servidor tendremos el siguiente método para gestionar los mensajes que van llegando al server.

oServerWebSocket:SetMessage(<bMessage>)

Donde <bMessage> será un codeblock a una función que recibe el mensaje del cliente. Aquí es donde empieza la primera pregunta, ¿qué mensaje? Pues esta parte será más un tema de diseño de cada uno imaginemos que tenemos una lógica que nuestro javascript puede enviar 3 mensajes que me invento: time, date, dummy.

Nuestra rutina de proceso de mensajes tendrá que tener una especie de gestor de mensajes.

oServerWebSockets:bMessages := { |cMsg| MyMessages(cMsg) }

Y la función podría ser algo así conceptualmente.

```
function MyMessages( cMessage )

      do case
         case cMessage == 'time' ; ProcessTime()
         case cMessage == 'date' ; ProcessDate()
         case cMessage == 'dummy'; ProcessDummy()
      endcase

retu nil
```

Cuando desde el servidor se procesa uno de estos mensajes, responde (o no) con otro mensaje que llega a nuestro cliente y javascript habría de saberlo también procesar como el que hemos definido en nuestro backend.

Pero si queremos aumentar aún más el grado de manejo de estos mensajes, podremos gestionarlos en función del scope que habremos definido si es así. Esto nos permitirá enviar/recibir mensajes de un módulo determinado sin interferir en otro, Por ejemplo, si tengo el módulo de ventas podría gestionar solo con este módulo los mensajes que queramos enviar.

Podemos crear otro concepto de diseño y nuestra función ya pasa a ser un “despachador” de mensajes, un “Dispatcher”. Esta función recibirá el mensaje y desde qué módulo se procesa.

Definimos nuestro despachador p.e de esta manera:

oServerWebSockets:bMessages := { |cMsg, cScope| MyDispatcher(cMsg, cScope) }

```
function MyDispatcher( cMessage, cScope )

      do case
         case cScope == 'ventas' ; ProcessVentas( cMsg )
         case cScope == 'chat'   ; ProcessChat( cMsg )
         ...
      endcase

retu nil
```

Y en cada función de módulo crear su casuística como p.e. el ejemplo anterior yMessages()

Este modelo ya nos permite escalar fácilmente nuestras diversas funcionalidades y manejo de los websockets. Creo que es una buena manera y fácil de implementar.

Al final podremos diseñar cómo queremos el manejo de los websockets.

Seguridad

Finalmente otro punto a tener en cuenta es la seguridad en el uso de los websockets. Si bien UT ya dispone de la capa de SSL para els uso del protocolo wss:// esto *solo* significa que la comunicación estará encriptada, pero hemos de tener en cuenta que quizás nuestra aplicación o módulo queramos que no sea de acceso libre (ni que esté encriptada la información vía SSL) sinó que queremos controlar este acceso.

Para conseguir esta funcionalidad, cuando nos autenticamos en nuestra aplicación podemos obtener un “token” que nos identifique y este token lo usaremos en la definición de nuestro websocket.

```
      DEFINE WEBSOCKET TOKEN 'ABC-1234'
```

Cuando definamos este token implica que nosotros habremos de crear una rutina que valide este token una vez nos conectemos, es decir cuando iniciamos el módulo de nuestra aplicación. Esta función ha de devolver un valor lógico .T./.F. que indicará simplemente si tiene acceso o no. Para indicar que rutina usaremos, una vez creemos en nuestra app el servidor de websocket, usaremos el método:

oServerWebSocket:SetValidate(<bBloque>)

Dónde <bBloque> será un codeblock de una función que recibirá como parámetro el token, el cual deberemos validar, sencillo.

Y que validamos ? Pues esto forma también parte del diseño, podemos por ejemplo al autenticar la entrada a nuestra aplicación recibir un token, JWT, … Y en cada módulo que necesitemos ejecutar websockets, cuando definamos los websockets definir este token como se ha comentado antes. Será la función que diseñemos para validar este ticket la que devuelva simplemente un valor lógico, autorizando o no el uso de los websockets.

Funciones y Métodos para manejo de WebSockets

Funciones Backend	**Descripción**
UWS_Send(<cSocket>, <uMsg>, <cScope>)	Enviar un mensaje al Frontend (Cliente) . ● <cSocket> → lo recibiremos a nuestra función y nos sirve para identificar el cliente. ● <uMsg> → Puede ser un string o un hash ● <cScope> Ambito de envio (opcional)
UWS_SendJS(<cSocket>, <cFunctionJS>, <uMsg>, <cScope>)	Igual que UWS_Send() pero el socket será enviado a la funcion Javascript <cFunctionJS>
UWS_GetInfoSockets()	Array de todas las conexiones persistentes. Cada elemento del array es un hash con las siguientes claves. ● token → Id. del token conectado ● ip → Ip del cliente ● module → scope del cliente ● in → Hora de entrada
UWS_ErrorHandler(oError)	Nos devuelve un hash con valores de error
UWS_InfoSSL()	Hash con información del certificado

Métodos oDom	**Descripción**
SendSocketJS(<cFunctionJS>, <uData>, <cScope>)	Enviar variable <uData> a la funcion javascript <uData> del cliente. Si se especifica el <cScope> enviara el mensaje a todos los

	socket conectados al ambito <cScope>
WSSetCargo(<u>)	Guarda la variable <u> en memoria asociada al puntero del socket
WSGetCargo()	Recupera la variable asociada al puntero del socket
UGetInfoSocket()	

Funciones Javascript	**Descripción**
UWS_Define(<hCfg>)	Configuración de la conexión websocket. hCfg es un hash con los siguientes valores. ● 'uri' → Direccion de conexión y puerto, ● 'scope' → Ambito de esa conexión, ex. 'CHAT' ● 'token' → Token que validará esa conexión ● 'onopen' → Funcion que se ejecutará en el evento onopen ● 'onclose' → Funcion que se ejecutará en el evento onclose ● 'onerror' → Funcion que se ejecutará en el evento onerror ● 'onmessage' → Funcion que se ejecutará en el evento onmessage ● 'console' → Si true, mostrará mensajes de sistema en la consola.
UWS_Init()	Crea una conexion Websocket. El sistema usará la configuración definida anteriormente. En el caso de que la conexion estuviera cerrada, algun error, …la función siempre reintentará la conexión en bas e a esa configuración
UWS_Send(<cMsg>)	Envia el mensaje <cMsg> al server
UWS_Close()	Cierra la conexión

Certificados

Se ha de tener muy claro el tema de los certificados y como usarlos. Cuando definamos nuestro sistema podemos hacerlo de 2 maneras: Con SSL o no y esto implicará usar certificado o no y unos puertos o no.

Como sabéis por defecto cuando definimos UT para web usamos el puerto 81 para que no interfiera con algún servicio que tengamos, pero que podemos servir el que queramos. En este caso si

trabajamos en local, iniciamos en el navegador como http://localhost:81 y para websockets podemos usar el 9000

En el caso de que activemos SSL, por defecto se usará el puerto 443 para web y podemos usar por ejemplo el 8883 para los websockets.

El motivo de no usar el mismo puerto, es que muchas veces puede dar errores y conflictos. Es por eso que recomiendo un puerto para cada server. Esto a la vez implica la habilitación en vuestro router y según como tengais configurado tambien el firewall.

También a tener en cuenta que no podéis usar por ejemplo un certificado para un dominio si lo queréis usar en localhost o viceversa

Cuando usemos websockets con SSL usaremos la librería \lib\uhttpd2\msvc64\uhttpd2.lib y hay un bat de ejemplo go_msvc64.bat

Si queremos usar SSL, tanto con un dominio o localhost usaremos la librería \lib\uhttpd2\msvc64\uhttpds2.lib y hay un bat de ejemplo go_msvc64_ssl.bat

La diferencia de usar un sistema u otro está en que el uso de SSL requiere de las librerías:

- libcrypto-1_1-x64.dll
- libssl-1_1-x64.dll

Codigo ejemplo

Ejemplo de configuración servidor sin SSL

```
//    ----------------------------------------------------------//
//    Config WebSockets Server
//    ----------------------------------------------------------//

function WebServerSocket()

      local oServer := UWebSocket()

      oServer:SetSSL( .F. )
      oServer:SetPort( 9000 )

      //    ------------------------------------------------------//
      IF ! oServer:Run()

            ? "=> WebServerSockets error:", oServer:cError

            RETU 1
      ENDIF

RETURN 0
```

Ejemplo de configuración con SSL

```
//    ----------------------------------------------------------------//
//    Config WebSockets Server SSL
//    ----------------------------------------------------------------//

function WebServerSocket()

      local oServer := UWebSocket()


      oServer:SetSSL( .T. )

      oServer:SetPort( 8883 )

      // You must specify the domain, not the IP.

      oServer:SetAddress( 'charles9000.work.gd' )

      // Certificate is for domain.
      // It won't work if you use it with localhost or other

      oServer:SetCertificate( 'cert/privatekey.pem', 'cert/certificate.pem' )

      //    ----------------------------------------------------------//

      IF ! oServer:Run()

            ? "=> WebServerSockets error:", oServer:cError

            RETU 1
      ENDIF

RETURN 0
```

Ejemplo FrontEnd – View

```
<?prg
#include "lib/tweb/tweb.ch"

  LOCAL o, oWeb

    DEFINE WEB oWeb TITLE 'UT Websockets'

      DEFINE FORM o ID 'myform' API 'api_websocket' OF oWeb

      INIT FORM o

        ROWGROUP o
```

```
          PROGRESS ID 'myprogress' VALUE 0 LABEL 'Status' PERCENTAGE GRID 12 ;
            CLASS 'bg-success progress-bar-striped progress-bar-animated' ;
            STYLE 'box-shadow: 5px 5px 5px gray;border: 1px solid #b1b1b1;'
HEIGHT 50 OF o

        ENDROW o

        ROWGROUP o HALIGN 'center'
          BUTTON ID 'btn' PROMPT 'Execute' ACTION 'my_process' GRID 12 WIDTH
'150px' ;
            CLASS 'btn btn-outline-dark' PBS 'InitProc' ALIGN 'center' OF o

        ENDROW o

      ENDFORM o

      DEFINE WEBSOCKETS TOKEN 'ABC-1234' OF o

      HTML o
        <script>

          function InitProc() {
            $('#myform-btn').html( '<i class="fas fa-spinner fa-
spin"></i> Executing...' )
            $('#myform-btn').prop("disabled",true);
            return true
          }

          function ShowProcess( data ) {

            $('#myform-myprogress_label').html( data['prompt' ] )
            $('#myform-myprogress').css({"width": data['bar' ] + '%' })
            $('#myform-myprogress').html( data['bar' ] + '%' )
          }

        </script>
      ENDTEXT

      INIT WEB oWeb RETURN
?>
```

Ejemplo Backend – API

```
#include 'hbsocket.ch'

// ------------------------------------------------------- //

function Api_Websocket( oDom )

  do case

    case oDom:GetProc() == 'my_process'    ; DoMy_Process( oDom )

  endcase
```

```
retu oDom:Send()

// ------------------------------------------------------- //

function DoMy_Process( oDom )

  LOCAL n, n100

  // Init long proc...

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Init process.
Working...', 'bar' => 1 } )
     SecondsSleep( 2 )

     n100 := 10

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Start first process',
'bar' => n100 } )
     SecondsSleep( 2 )

     n100 += 30

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'First process finished.
Preparing reporting...', 'bar' => n100 } )
     SecondsSleep( 2 )

     for n := 1 to 20

       n100++
       oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Generating reporting '
+ ltrim(str(n)), 'bar' => n100 } )
       SecondsSleep( 0.1 )

     next

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Start second process',
'bar' => n100 } )
     SecondsSleep( 2 )


     n100 += 20

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Second process
finished...', 'bar' => n100 } )
     SecondsSleep( 2 )

     for n := 1 to 20

       n100++
       oDom:SendSocketJS( 'ShowProcess', { 'prompt' => 'Preparing output ' +
ltrim(str(n)), 'bar' => n100 } )
       SecondsSleep( 0.1 )
     next

     oDom:SendSocketJS( 'ShowProcess', { 'prompt' => '<h4><b>Process finished
successfull !</b></h4>', 'bar' => 100 } )
```

```
  // End Proc ----------------------

     oDom:Set( 'btn', 'Execute' )
     oDom:Enable( 'btn' )

return nil

// ------------------------------------------------------------ //
```

Resumen

No hay lectura como en Harbour crear un sistema de estas características, pero este es un prototipo completamente funcional y una manera de gestionarlo tal como entiendo que se habría de hacer desde UT siguiendo su filosofía y como veis es todo un ecosistema que tendremos que entender para poderlo gestionar correctamente.

Podemos fácilmente ver que simplemente definiendo los websockets cuando definimos la pantalla por un lado y definir en nuestro api como queremos responder conseguimos nuestro objetivo. Por otra parte, a un nivel más libre somos nosotros quien podemos gestionar todos los mensajes de un lado para otro

Frontend ←→ Backend

Ha sido un nuevo reto poder conseguir este objetivo pero ya tenemos una funcionalidad que nos dará muchas opciones de sumar más capacidad a nuestro módulo.

Saltamos a **UT 2.0**