Chat Integration in 4D v11 SQL

By Atanas Atanassov, Technical Services Team Member, 4D Inc.

Technical Note 09-44

Table of Contents

Abstract

Being able to communicate with users at other workstations is important in many businesses. Having chat functionality integrated in a developer's application can thus be very beneficial. Users can exchange critical information such as what work they are doing in the application or even just say hello. 4D Server has a built in feature for sending messages to clients, but it is not meant to be used like a chat application. Fortunately, chat functionality can be built fairly easily with 4D v11 SQL.

Introduction

4D Server v11 SQL can send messages to clients with important information from the server administrator. Clients connected to the server can send replies to the server and if they have Designer or Administrator rights. Sometimes it is important for users to send short messages to other users. This can easily be done in 4D.

This Technical Note explains how to integrate chat functionality in 4D. The chat area is built with 4D's Web Area. This gives developers an easy solution to create text areas for displaying typed messages. Using the Web Area approach gives developers the ability to use JavaScript and CSS to customize the view and the feel of the messages.

An explanation of the model used to build the chat application is introduced first then continues with building the chat UI and explanation of the message flow between clients through the server.

A sample database is included that demonstrates the approach used for building the chat application.

Requirements

The minimum requirements are 4D Server v11 SQL Release 5 and two 4D v11 SQL Release 5 applications connected to the server.

Explanation of the Model

When a client connects to the server, it is registered with a unique name and it is visible for all other clients connected to the server. Other clients can start a chat with this client and vice versa. For simplicity we name the client that initiates the chat as the "Caller" client and the requested client as the "Called" client.

Once the Caller client opens a chat session with the Called client, it sends a message which is stored temporary on the server. During the chat initiation process on both clients, a process is launched that periodically checks for sent messages. If there are any, the process gets and delivers them to the right client. Messages that

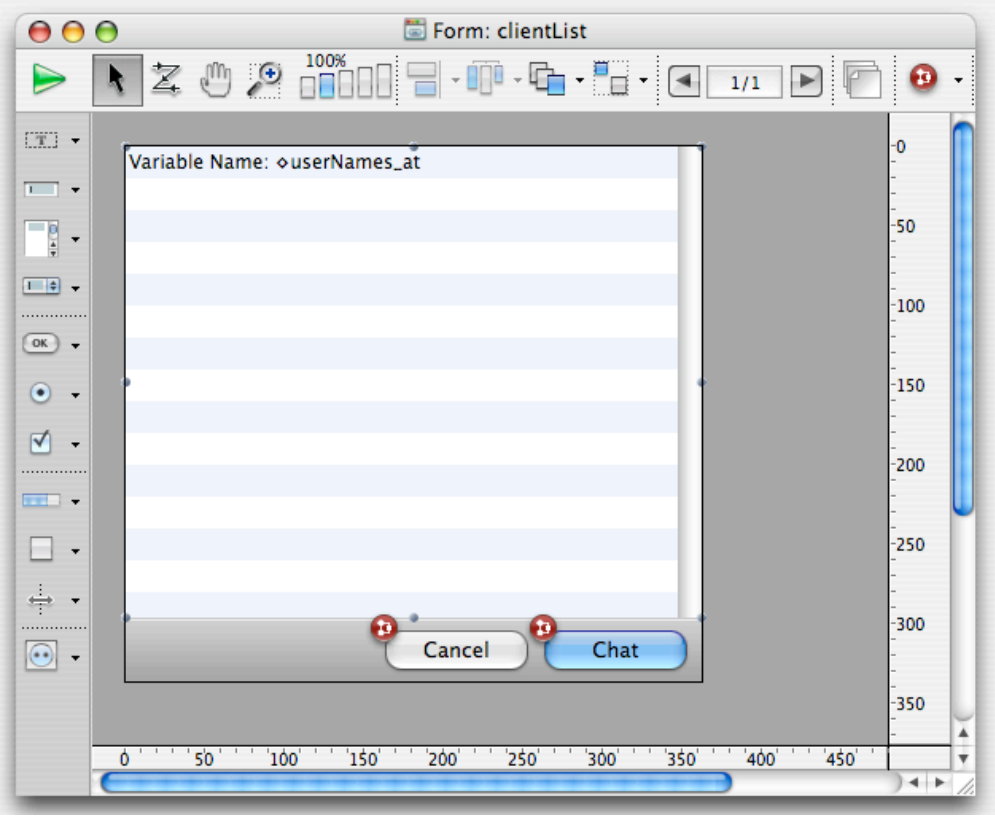
are sent to the server have their status set to false-the message is not read yet. Once it is read, the status is changed to true. The process of getting messages grabs only the unread messages.

Implementation

The implementation starts with creating the forms to list all available clients and the chat UI.

List of Available Clients Form

The form has a List box based on arrays and two buttons “Cancel” and “Chat”



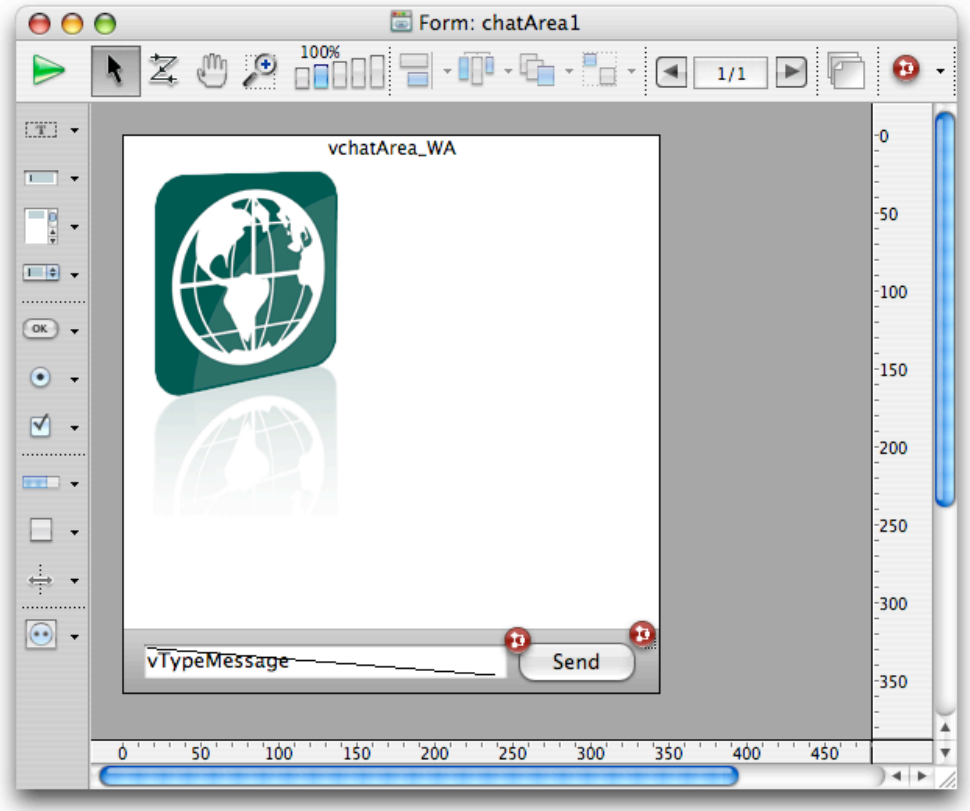
The names of all registered clients are saved in the array named “<>userNames_at”.

The “Chat” button loads the chat window/area. Every chat window starts in its own unique process. Thus clients can initiate multiple chat sessions.

Chat area

The chat form contains a Web Area that displays the ongoing conversations between two clients. Every chat has its own chat area. The Web Area allows

users to customize the appearance and text flow of all messages by using CSS and JavaScript.



The html page called "Chat.html" is in the Resources folder and is available to all clients connected to the server. The contents of this file are displayed in the Web Area and it is updated dynamically during the conversation. Every message is placed in a separate "div" tag and is added to the bottom of the page.

Note: *"Div" tags define a division or a section in an HTML document. They are often used to group block-elements to format with styles.*

Let's look at the code in Chat.html

The first elements in the head tag are styles. The style tag contains the definition of all CSS for the html page:

```
<style type="text/css">
                .send
                {
                        /* style for the sent message */
                        background-color:#E0FFFF;
                        text-align:left;
                        font-family: Verdana;
                        font-size:12px;
                        padding: 5px;
                }
```

```
                .received
                {
                       /* style for the received message */
                       background-color:#FFF8C6;
                       text-align:right;
                       font-family: Verdana;
                       font-size:12px;
                       padding: 5px;
                }

                .text {text-align:center;} /* header text */
                p {margin:0;} /* text in p tags */
                div {margin-bottom:5;} /* distance between the div tags */

                .messageSend
                {
                    /* style for the send header */
                       background-color:#E0FFFF;
                       font-style:italic;
                       font-size:9px;
                       text-align:center;
                       font-family:Arial;
                       padding: 5px;
                }

                .messageReceived
                {
                    /* style for the receive header */
                       background-color:#FFF8C6;
                       font-style:italic;
                       font-size:9px;
                       text-align:center;
                       font-family:Tahoma;
                       padding: 5px;
                }

                .span
                {
                    /* style for the text inside p tags */
                       margin-bottom:6;
                       margin-top:6;
                       margin-left:5;
                       margin-right:5;
                }
</style>
```

"Send" and "receive" classes customize the view of sent and received messages. They are formatted differently to help users quickly distinguish between sent and received messages.

The message header and body are separated in different paragraphs. Thus we have different styles for each of them. The classes "messageSend", "messageReceived" and "text" control the appearance of message headers.

The JavaSctip function called addChart controls the dynamic representation of all elements in the web area.

```
   <script type="text/javascript">
   function addChat(textToAdd, action, client){
```

```
        var div = document.getElementById("MainChatDiv");
        var p, text;
        var chatMessage;

        dv = document.createElement('div');
        document.body.appendChild(dv);
        p = document.createElement('P');
        p1 = document.createElement('P');
        span = document.createElement('span');

        if(action == 'send'){
              p.className = 'send';
              p1.className ='messageSend';
        }else if(action == 'received'){
              p1.className ='messageReceived';
              p.className = 'received';
        }

        chatMessage = 'Message from ';
        chatMessage +=client;
        chatMessage +=' ';

        var currentTime = new Date();
        var hours = currentTime.getHours();
        var minutes = currentTime.getMinutes();

          if (minutes < 10){
               minutes = "0" + minutes
          }

           var hoursShort = hours % 12;
           chatMessage += hoursShort+':'+minutes;

           if(hours > 11){
                 chatMessage +='pm';
           } else {
                 chatMessage +='am';
           }

           myText1= document.createTextNode(chatMessage);
           text= document.createTextNode(textToAdd);
           span.className = 'span';
           span.appendChild(text);
           p1.appendChild(myText1);
           p.appendChild(span);
           dv.appendChild(p1);
           dv.appendChild(p);
           window.scrollTo(0,10000);
        }

  </script>
```

Client side implementation

Every client is registered with a unique name during the connection to 4D Server. The registered client name is a combination based on the machine name and the user name. For example: StevenJones' XP_Designer. In this example the name of the machine is StevenJones' Xp and the user is logged as a designer.

Note: *Multiple clients can use one and the same machine to connect to 4D Server.*

Once the Called client is selected from the list of registered clients and the chat is initiated, the "On Load" form event loads the preferences and content of the Web Area. When the message is typed, first it is displayed in the Web Area by using ***WA EXECUTE JAVASCRIPT FUNCTION*** and is sent to the server. Displaying the message and sending to the server is handled in the ***Ch_EnterKey*** project method.

```
C_TEXT($processName_t)
C_LONGINT($processState_l)
C_INTEGER($processTime_h)

PROCESS PROPERTIES(Current
process;$processName_t;$processState_l;$processTime_h)
requestedClient_t:=Replace string($processName_t;"ChatWith_";"")
WA EXECUTE JAVASCRIPT FUNCTION(vchatArea_WA;"addChat";*;vTypeMessage;
"send";UTIL_EscapeString (<>clientId_t;"'"))
Ch_sendText (<>clientId_t;"sendMessage";requestedClient_t;vTypeMessage)
vTypeMessage:=""
```

This method first gets the process name for the ongoing conversation, displays the sent message in the Web Area and sends the message to the server.

The ***Ch_checkMessages*** project method frequently checks for messages left on the server for the Caller and Called clients. The "sendText_b" variable contains all messages for a requested client, either Caller or Called. It is a blob variable containing an array with messages. We use an array because several messages can be left on the server for a client. Here is the code for this method.

```
C_TEXT($message_t;$client_t)
C_LONGINT($index_l;$clientPos_l)
C_BLOB(sendText_b)
ARRAY TEXT($messageToReceive_at;0)

sendText_b:=Ch_getText (requestedClient_t)

  `Getting the name of the requested client
  `------------------------------------------------------------------
$clientPos_l:=Position (<>clientId_t;requestedClient_t)
$client_t:=Substring(requestedClient_t;1;$clientPos_l-2)
  `------------------------------------------------------------------
BLOB TO VARIABLE(sendText_b;$messageToReceive_at)
If (Size of array ($messageToReceive_at)>0)
   For ($index_l;1;Size of array($messageToReceive_at))
     WA EXECUTE JAVASCRIPT FUNCTION(vchatArea_WA;"addChat";*;
$messageToReceive_at{$index_l};"received";UTIL_EscapeString ($client_t;"'"))
   End for
End if
```

Server Side Implementation

Ch_sendText and ***Ch_getText*** project methods are executed on the Server. Both methods have the "Execute on Server" property checked.

The ***Ch_sendText*** method populates the message, status of the message and the ongoing chat information into the <> message_at_at, <>message_read_ab_ab and <>chatClients_at arrays. These three arrays are initialized upon Server start. <>message_read_ab_ab is a Boolean array that holds the status of the sent messages. When a message is sent to the server its status is set to False, and when it is sent to the requested client, the status of the message is set to True. This array and the <>message_at_at array are two dimensional arrays. This is a convenient way to store all conversations between the clients. The arrays are synchronized based on the <>chatClients_at array that contains the information for all ongoing conversations. The first chat is saved in the first element and this is the first row for the <>message_at_at and <>messages_read_ab_ab arrays.

```
$clientCaller_t:=$1
$action_t:=$2
$clientCalled_t:=$3
$messge_t:=$4

$chat_t:=$clientCaller_t+"-"+$clientCalled_t

  `filling the elements of message_a and message_read arrays

$chatFound_l:=Find in array(<>chatClients_at;$chat_t)

If ($chatFound_l=-1)  `chat is already started

   <>colIndex_i:=1
   $sizeChatClient_l:=Size of array(<>chatClients_at)
   <>rowIndex_i:=$sizeChatClient_l+1
   INSERT IN ARRAY(<>chatClients_at;<>rowIndex_i;1)
   <>chatClients_at{<>rowIndex_i}:=$chat_t
   INSERT IN ARRAY(<>message_at_at;<>rowIndex_i;1)
   INSERT IN ARRAY(<>message_read_ab_ab;<>rowIndex_i;1)

Else   `chat not found

   $arrayColSize_l:=Size of array(<>message_at_at{$chatFound_l})
   <>rowIndex_i:=$chatFound_l
   <>colIndex_i:=$arrayColSize_l+1

End if

Case of
   : ($action_t="sendMessage")  `sending message

         INSERT IN ARRAY(<>message_at_at{<>rowIndex_i};<>colIndex_i;1)
         INSERT IN ARRAY(<>message_read_ab_ab{<>rowIndex_i};<>colIndex_i;1)
         <>message_at_at{<>rowIndex_i}{<>colIndex_i}:=$messge_t
         <>message_read_ab_ab{<>rowIndex_i}{<>colIndex_i}:=False

   `Check to see if the chat window is still opened on the client machine

         EXECUTE ON
CLIENT($clientCalled_t;"Ch_P_startSession";$clientCaller_t)
End case
```

The ***Execute on client*** command is used to launch the chat window for the requested client.

The ***Ch_getText*** project method is called based on the "On Timer" form event and checks the see if there are unread messages for the client. All unread messages are saved in the array named $messageToSend_at. There is rarely more than one message to send because the timer is set to 0 ticks. Having more than one message to send can happen if the timing is set to bigger intervals. The array is saved in a blob variable and this variable is returned to the client.

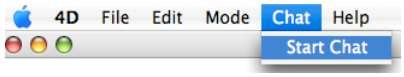
```
ARRAY TEXT($messageToSend_at;0)
C_BLOB(sendText_b;$0)
C_LONGINT($arrayIndex_l;$messageIndex_l)
C_LONGINT($foundFalse_l;$foundat_l)
C_TEXT($clientId_t;$1)

$arrRow_l:=Size of array(<>chatClients_at)
$clientId_t:=$1
$foundat_l:=Find in array(<>chatClients_at;$clientId_t)

If ($foundat_l#-1)
   Repeat
     $foundFalse_l:=Find in
array(<>message_read_ab_ab{$foundat_l};False;$foundFalse_l+1)
     If ($foundFalse_l#-1)
          APPEND TO
ARRAY($messageToSend_at;<>message_at_at{$foundat_l}{$foundFalse_l})
                  <>message_read_ab_ab{$foundat_l}{$foundFalse_l}:=True
     End if
   Until ($foundFalse_l=-1)
End if
VARIABLE TO BLOB($messageToSend_at;sendText_b)
$0:=sendText_b
```

When the chat window is closed, all messages for this chat are deleted from the server. Message deletion is handled in the ***Ch_DeleteChat*** project method.

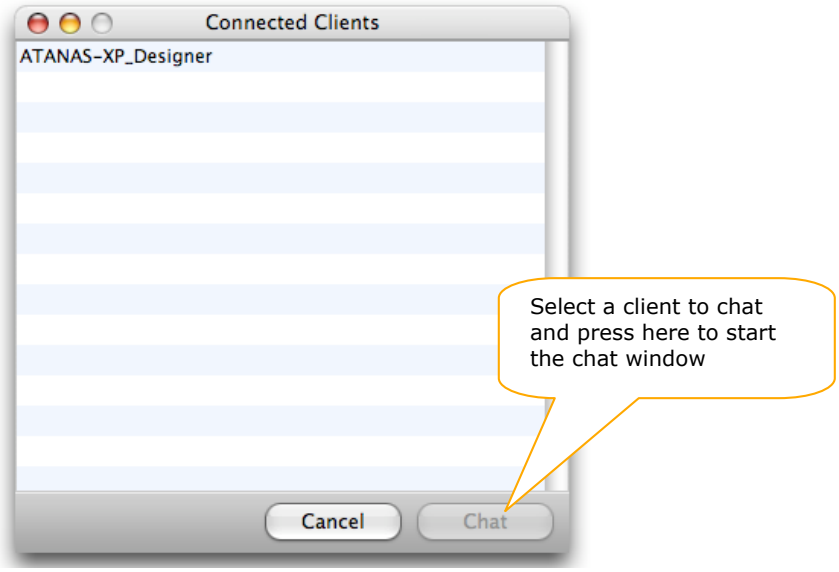
Demo Database

The included sample database, Chat.4dbase, demonstrates the implementation of a chat application into 4D as described above.

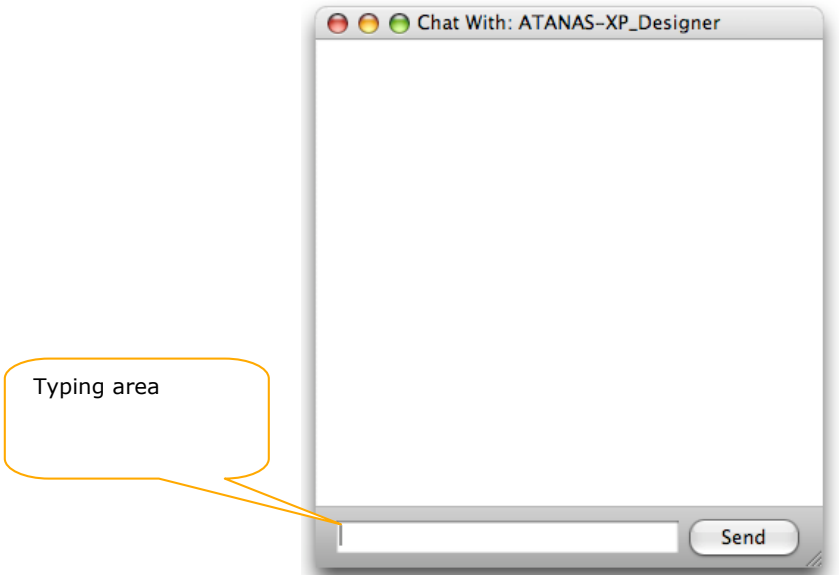
Launch the database with 4D Server v11 SQL. Connect at least two clients to the server. From the client select the Start Chat option from the Chat menu.



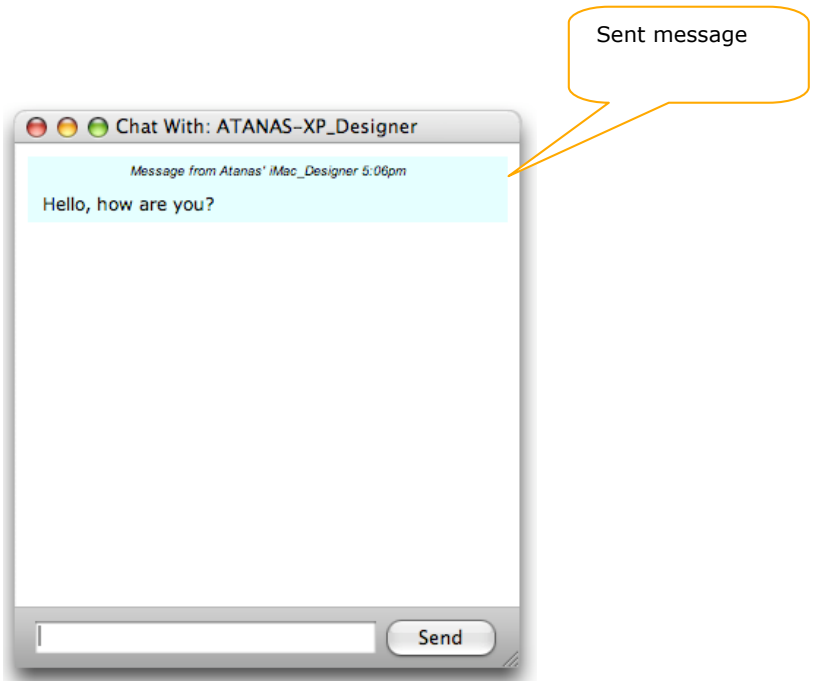
The "Connected client" form is called with the ***Ch_P_startChat*** project method and it loads the list of available clients to talk to.



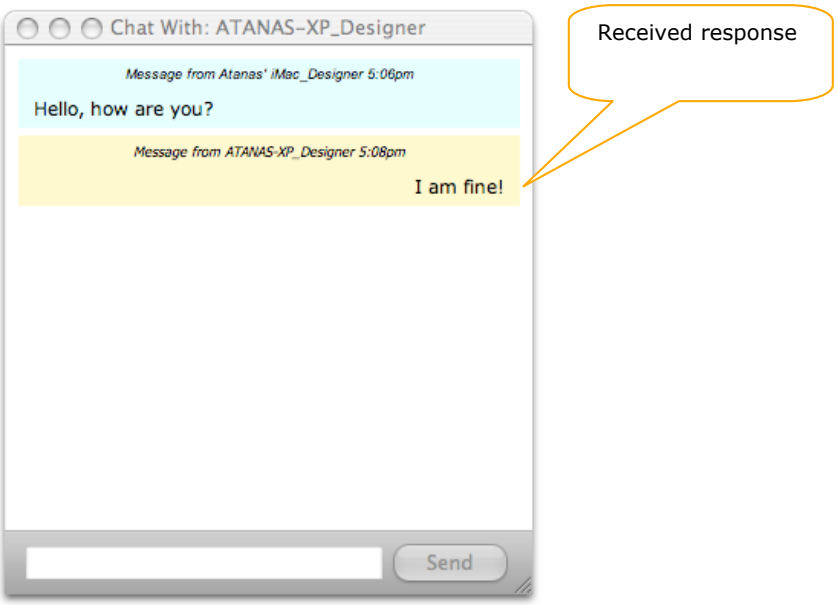
Select one of the available clients and click the Chat button. This loads the chat window.



Type a message in the typing area and press Send or hit the Return or Enter keys. The message is displayed in the area above and sent to the server.



From the server, it is then sent to the requested client and displayed in the chat window opened on the client screen.



Conclusion

4D v11 SQL has the ability to build applications similar in functionality to ones we use almost every day to accomplish different tasks such as chatting. The Chat application is such an example. It accomplishes the same task as popular chat clients like AIM and Google Talk. The difference is it is integrated in the database and can be tailored to users requirements. 4D Developers can use the code from this Technical Note to extend or modify its capabilities, functionalities and look in their custom chat applications.