

4th Quarter SOAP Application Program Interface

Lincoln Stoller

August 1, 2011

Contents

Introduction

Soap is Similar to 4D Open	2
Synchronous vs. Asynchronous	3
Single Proxy Method	4

Client-Side Issues

Issuing Soap Call	4
On Err Call Method	5
Testing The Connection	6

Input and Output Parameters

The Action Parameter	7
The Blob Parameter	8
Return Parameters	8

Monitoring the Server

The Synchronous Case	10
The Asynchronous Case	11
The Batch_Status Action	11

Server-Side Issues

Starting Web Services	14
The Timeout Issue	15
Errors on the Server	17

Summary

Soap Client Sample Code	17
-------------------------	----

Introduction

Soap is Similar to 4D Open

4th Quarter's Soap inter-application protocol is based on its previous implementation of 4D Open. 4Q's Soap API supports the same series of calls and requires the same data, formatted in the same way, as the 4D Open API.

Users with a source code license to 4th Quarter have access to the SOAP/4D Open APIs. 4Q users with runtime licenses to 4Q Core and 4Q Business must have purchased the additional license to run these APIs. Without this additional license, which may be purchased as an upgrade, 4th Quarter will not respond to API calls. A list of all 4Q's API calls is shown in the following section titled "Action Parameter".

The main feature of 4th Quarter's 4D Open, and now also of Soap, is a middle-ware layer in 4th Quarter that logs all requests to update the database using separate records in the [Batch_Task] table.

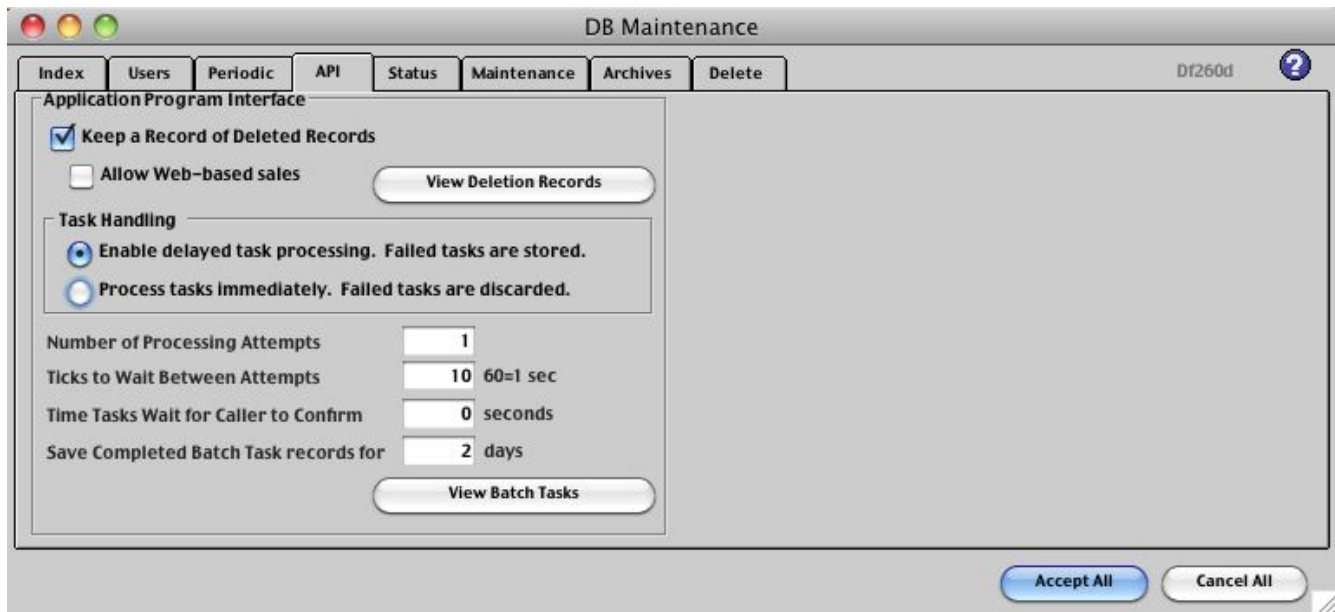
Batch_Task records are created when 4D Open or Soap calls are received that request the addition, modification, or deletion of data. These records record the time and source of the request, all the data provided in the request, they log the ongoing processing of the request, and they ultimately record the final outcome. In addition, the 4th Quarter Administrator can resubmit well-formatted requests that were not able to complete successfully due to issues of data contention. Administrator access to these records is through the Application Program Interface pane of the DB Maintenance screen, which it accessed from the File menu accessible from 4Q's control screen.

On this page you'll find access to the following settings pertinent to Soap calls:

Number of Processing Attempts	Number of attempts the server will make to access locked records before it abandons the request.
Ticks to Wait Between Attempts	Number of ticks between each attempt to access locked records.
Save Completed Batch Task Records	Number of days that records of successful operations will be retained. Successful records are not actually deleted once they age beyond this time, but they are marked for reuse as records for subsequent Soap calls

The check box labeled "Keep a Record of Deleted Records" does not apply to Soap calls because record of Soap-initiated record deletion is always kept.

To access Batch Task records in order to review or delete them, press the **View Batch Tasks** button shown below. This opens a list screen for searching, examining, deleting, and re-running previous Soap requests stored as Batch Task records. Batch Task records cannot be created by the Administrator; they can only be created by Soap-based client requests.



The 4D Open protocol for communications between 4th Quarter and other 4D applications is detailed in our document titled 4Q API Manual that is available on the 4th Quarter website at www.4thquarter.com/DownloadF/4Qdocuments/4QAPIManual.pdf. Refer to this manual for complete details of data formats required to implement each of the Soap or 4D Open calls, and for further description of Batch Task records.

Synchronous vs. Asynchronous

The difference between the Soap and 4D Open API lies in the communication between the client and the server. The 4D Open server executed instructions asynchronously. The client had to implement a mechanism for waiting for the server to complete its task if the client wanted to know the outcome of the operation. In contrast, the Soap API all client execution halts until the server completes the action requested. For the duration of the Soap operation the client is synchronized with the server. For this reason Soap is known as a synchronous communication protocol.

Synchronous protocols, like HTTP and SMTP, are suited for data transfer but are less appropriate for the time consuming tasks of modifying data. For this reason the 4th Quarter Soap API is a mixture of synchronous and asynchronous commands. 4Q's Soap commands are synchronous when they are strictly requests for data from the server, and they are asynchronous whenever they constitute a request to modify data on the server. All information retrieval requests are synchronous; all data updating requests are asynchronous.

Synchronous processing is simpler as nothing can happen on the client until the action is complete, and once the client recommences execution operations on the server are unequivocally over. Asynchronous processing leaves the client free to continue other operations but also in the dark as to the outcome of the request. 4Q's asynchronous calls require the client check back later, using the 4D Soap "Status" call, to determine if the action has completed successfully or not, or if processing is still ongoing. Monitoring the result of an action is described in the section titled "Monitoring the Server."

Single Proxy Method

4th Quarter's Soap protocol consists of a single proxy method. This method is executed by the 4th Quarter application acting as the Soap server. The client calls this method using the Soap API and passes this method an action parameter that describes the action to be undertaken. The client also passes a Blob object containing the data needed to complete this action. The client can be any SOAP-enabled application but for the purposes of providing code examples we will assume the client is a 4D application.

All Soap calls are calls to this single proxy method. This is true whether the client is requesting information from the server, passing information to the server, or querying the status of a previous command issued to the server.

Client-Side Issues

Issuing Soap Call

The following assumes the client is executing its call from a 4D-based environment. The following examples use 4D code.

Prepare the variables needed for the Soap call by assigning values to the BatchAction and, optionally, DataBlob named parameters. The BatchAction variable is a text variable whose contents tell the server what action to perform. The DataBlob is a blob-type variable that contains the data needed for this action. The following example assumes that you have sized and filled a text array named vArrayText with the data the server needs to complete the action that you have assigned to the vWSBatchAction text variable.

In the following code you need to replace the IP address shown as 10.0.1.6 with the correct IP address for your Soap server.

```
C_TEXT( vWSBatchAction; vWSReturnText; vWSReturnCode)
• C_LONGINT( $0; $ReturnErr)
C_BLOB(vWSReturnBlob)

SET BLOB SIZE(vWSDataBlob; 0)
vSoapError := False
$ReturnErr := 0
vWSBatchAction:="List_All"
VARIABLE TO BLOB(vArrayText; vWSDataBlob)
SET WEB SERVICE PARAMETER("BatchAction"; vWSBatchAction)
SET WEB SERVICE PARAMETER("DataBlob"; vWSDataBlob)

$SOAPServerAddress :=http://10.0.1.6:8080/4DSOAP
$SoapAction := "WebService4q#_BTProxy4QSubmitTask"
$Method := "_BTProxy4QSubmitTask"
$NameSpace := "http://www.4q.com/namespace4q/default"

ON ERR CALL("YourWebServiceErrorHandler")
```

CALL WEB SERVICE(\$SOAPServerAddress; \$SoapAction; \$Method; \$NameSpace;
Web Service Dynamic)

If (OK=0)

 `Call failed or Server was not located. Handle your error here.

Else

GET WEB SERVICE RESULT(\$0; "ErrorCode")

GET WEB SERVICE RESULT(vWSReturnCode; "ReturnCode")

GET WEB SERVICE RESULT(vWSReturnText; "ReturnText")

GET WEB SERVICE RESULT(vWSReturnBlob; "ReturnBlob")

 \$ReturnErr := \$0

End If

ON ERR CALL("")

If(\$ReturnErr = 0)

 `Process completed correctly or is ongoing. Handle monitoring operations here.

 `See following sections titled "When to Monitor The Server".

Else

 `Process did not complete as expected. Handle your error here.

End If

On Err Call Method

You need to create and install an On Err Call method to intercept web service related errors. This is done in the usual manner of installing any On Err Call method using the On Err Call command. This error handling method should be deinstalled only after all web service operations are complete.

4D provides two data items when a web service action fails. These are retrieved using the Get Web Service error info method. A simple error handling method is as follows:

 `On Err Call method for Web Service errors

 \$ErrMsg := **Get Web Service error info** (Web Service Detailed Message)

 \$ErrSource := **Get Web Service error info** (Web Service Fault Actor)

 vSoapError := **True**

 vErrorDescription := \$ErrMsg + \$ErrSource

The Web Service Fault Actor parameter results in the return of either the value "client fault" or "server fault". Unfortunately the server fault error descriptions returned by 4D are of little use in locating the source of the error. Here are a few of the error codes I have received during development along with the actual cause of the error as I was subsequently able to determine.

Error number	Actual cause
16	Making an assignment using an undefined local variable.
16	Attempting to assign a numeric value to a string variable.
53	Attempting to access a nonexisting array element in a process array variable.
55	Testing an undefined process variable in an If statement in the proxy method.
61	Attempting to extract a longint value from a Blob containing a text array.

I have also received the message “Permission error. Attempt to open locked file for writing” when, in fact, the error was nothing of the sort. Three different errors that generate this same misleading message were dereferencing a nil pointer, dereferencing an undefined method parameter, and attempting to assign a passed parameter of one type to a local variable of another type.

Testing The Connection

Before calling the Soap server for an application specific request make sure that the Soap server can be found and will respond. This can be done in two ways. First, during development, you can request the WSDL from the server by typing the following URL into the target field of any browser:

```
http://10.0.1.6:8080/4DWSDL
```

In this call use your own IP address in place of the value 10.0.1.6 and, if you’re using a different port number for the 4Q server, use the appropriate value in place of 8080.

If you have launched the server application and have turned Web Services on, and you have a valid Web Services license for the application, then the browser should return the WSDL in unformatted XML text. If the server has not been launched, or if you have not turn on web services, then you’ll get a message telling you that the server cannot be found, or something to that effect.

If you don’t have a web services license then you can still use any 4D development or server application as a web server for a 1 hour trial period following each launch. This still requires that you start web services in the Web pane of the 4D Preferences dialog.

Once you’ve established a reliable protocol for making the connection to the server you can issue the following call by assigning the value SOAP_TEST to the vWSBatchAction variable in the above code block:

```
vWSBatchAction := “SOAP_TEST”
```

This call is handled by the server assigning the following values in \$0 and the named parameters:

```
$0 = 1  
ReturnCode = “1”  
ReturnText = “Simple test return text OK.”  
ReturnBlob = contains a 2-element, 1 dimensional text array.
```

The Blob’s text array can be extracted using the Blob To Variable command. The contents of the elements of this text array are as follows:

```
{0} = 2  
{1} = “Test_Element_1”  
{2} = “Test_Element_2”
```

Input and Output Parameters

The Action Parameter

The action requested by the client is indicated by a text value that's placed in a Soap parameter named BatchAction. Every client request must provide a recognizable string value. This string consists of one or more substrings each of which must exactly match recognized strings on the server, although the case of the characters is unimportant. In most cases where the an incorrect string is passed the server will take no action and return an error code that indicates it failed to recognize the action requested. In rare cases the server could recognize parts of the command and undertake an action different than the client intended.

The following is a list of the actions recognized by the server. The list of actions in the left column are available to all 4Q users of the Core application who have licensed the 4Q APIs, or users of the Core application who have a source code license. The list in the right column are the additional calls available to 4Q users of the 4Q Business application who have also licensed the APIs, or who have a license to the source code of the 4Q Business application.

4Q Core API Calls

SOAP_TEST
LIST_ALL_ACTIONS
BATCH_STATUS
BATCH_DELETE
ACCOUNT_NEW_CHECK
ACCOUNT_MODIFY_CHECK
ACCOUNT_NEW
ACCOUNT_MODIFY
ACCOUNT_DELETE
ACCOUNT_DESCRIPTION
TRANSACTION_NEW_CHECK
TRANSACTION_MODIFY_CHECK
TRANSACTION_MODIFY
TRANSACTION_NEW
TRANSACTION_DELETE
TRANSACTION_DESCRIPTION

Additional 4Q Business API Calls

CUSTOMER_NEW_CHECK
CUSTOMER_MODIFY_CHECK
CUSTOMER_NEW
CUSTOMER_MODIFY
CUSTOMER_DELETE
CUSTOMER_DESCRIPTION
CUSTOMER_NEW_RECEIVABLE_ACCT
INVOICE_NEW_CHECK
INVOICE_MODIFY_CHECK
INVOICE_NEW
INVOICE_MODIFY
INVOICE_DELETE
INVOICE_DESCRIPTION
VENDOR_NEW_CHECK
VENDOR_MODIFY_CHECK
VENDOR_NEW
VENDOR_MODIFY
VENDOR_DELETE
VENDOR_DESCRIPTION
PURCHASE_NEW_CHECK
PURCHASE_MODIFY_CHECK
PURCHASE_NEW
PURCHASE_MODIFY
PURCHASE_DELETE
PURCHASE_DESCRIPTION

These action commands adhere to the following rules. Actions commands that request the storage of a new record on the server contain the substring “_New”, all actions that request the modification of a record contain the substring “_Modify”, and all actions requesting deletion record contain the substring “_Delete”. Commands that contain the substring “_Check” are interpreted as requests to check the format and contents of data provided in the Blob parameter.

The main area of the 4Q application to which these changes apply are identified by the substrings “_Vendor”, “_Customer”, “_Account”, and so forth. Actions with the substring “Invoice” or “Purchase” apply to invoice or purchase order areas respectively. “Purchase” does not refer to purchase-type transactions, which are handled as using parameters that apply to the handling of transactions.

Some actions neither modify data nor apply to specific files in the 4Q database. “List_All” requests the return of a list of all recognized actions. “Status” requests the return of the status of a previous Soap request, an request that is identified by an additional parameter that's passed in the Blob parameter that's also passed.

The Blob Parameter

Each call must be accompanied by a Blob that is referred to by the name “DataBlob”. This Blob must be passed even if it provides no data, in which case the Blob can be empty. That is to say even if the Blob is unused it must still be passed.

The Blob contains two kinds of information: the target of the action, and the data needed to complete the action. This can be as little as a single number, such at the number that identifies a previous action for the purposes of determining if that task has completed, or it may contain all the information needed to create a parent and its related children records.

The client places information in the Blob using 4D's Variable to Blob or Array to Blob commands. Refer to the 4D Open manual for the data formats required for different actions. Soap requests originating from other than 4D clients must format the Blob in a manner such that the server can extract this information using the Blob to Variable, or Blob to Array commands.

Return Parameters

All Soap calls to the Server's one proxy method return an longint error code and data in three named parameters. The longint error code is returned in \$0, and the named parameters are “ReturnCode”, “ReturnText”, and “ReturnBlob”. The contents returned in the named parameters differs between different calls, but the error code in \$0 is always returned and has the same meaning in all cases. When the value 0 is returned in \$0 this indicate that the requested action was understood and either was or is being processed.

If the requested action executes synchronously, then a value equal to or greater than 0 returned in \$0 indicates the request was processed successfully. A value less than zero indicates an error either in the syntax of the request, or the execution of it. In either case the task is complete.

If the requested action executes asynchronously, then a value of 0 in \$0 indicates the request was understood and the server is proceeding to examine the data and fulfill the request. In this case a reference number is also returned in the text parameter named “ReturnCode.” This reference number can be used in a subsequent “Status” call to learn of the ongoing state of this request on the server. As long as \$0=0 and the value in the ReturnCode variable is greater than zero, then the task is being or has been processed in another 4D process on the server. If the value in \$0 or the ReturnCode variable is less than zero, then the client's action request was not accepted and no additional processing occurred on the server.

Monitoring the Server

The sever returns four variables that contain data: the \$0 method parameter, and the three named variables “ReturnCode”, “ReturnText”, which are of type text, and “ReturnBlob”, which is of type Blob. In general, these values are used for the following purposes:

\$0	Longint	Returns an error code. A negative value indicates an error, while a zero or positive value indicates no error. A positive value generally provides a reference number the client can use to inquire about the status of a task that is being processed asynchronously.
ReturnCode	Text	Return a code that is usually equal to that returned in \$0 but may differ in some cases. In the case of record creation and modification actions that run asynchronously the value returned in ReturnCode is the number of the process in which the task is running, while \$0 returns the ID number of the Batch Task needed when inquiring about the process using the Batch_Status command.
ReturnText	Text	Returns a readable text description of the result of the requested action. If there has been an error, then this description will describe it. If the action has completed successfully, then a value of the following form is returned: “OK/##/Action Description. That is, it will begin with the word “OK” followed by a slash, the number of attempts that were made in order to successfully process the request, another slash, and then a description of the task accomplished.
ReturnBlob	Blob	Returns data requested by the client. This data is generally returned in the form of a text array whose elements can be accessed on the client by issuing the Blob To Array command. Arrays are returned in the following cases:

The following actions return data in the ReturnBlob variable.

Action value	Additional Input	Return Values
BATCH_STATUS	Task ID in first element of text array packed into DataBlob	\$0 ReturnBlob containing single array of status values.
ACCOUNT_DESCRIPTION `	none	\$0 ReturnBlob containing single array of input account field names.
TRANSACTION_DESCRIPTION	none	\$0

		ReturnBlob containing two arrays, the first holding transaction header field names, the second holding transaction component field names.
CUSTOMER_DESCRIPTION	none	\$0 ReturnBlob containing single array of customer field names.
INVOICE_DESCRIPTION	none	\$0 ReturnBlob containing two arrays, the first holding invoice header field names, the second holding line item field names.
VENDOR_DESCRIPTION	none	\$0 ReturnBlob containing single array of vendor field names.
PURCHASE_DESCRIPTION	none	\$0 ReturnBlob containing two arrays, the first holding PO header field names, the second line item field names.

The Synchronous Case

Synchronous calls do not require monitoring of the server since there is no further processing on the server once a synchronous call returns execution control to the client. The only post-processing information needed or available in this case is in the value of \$0 and the named parameters.

Any action string passed to the Server's `_BTPProxy4QSubmitTask` method that does not contain the string `"_NEW"` or `"_MODIFY"` is handled synchronously. This includes requests to delete data, which are handled synchronously.

The general rule, which is usually sufficient, is that if the value of \$0 is 0, then the task completed successfully. In this case the value of ReturnCode reiterates the value in \$0, the value in ReturnText describes the action that was completed, and the value in ReturnBlob contains the data that was requested.

When an error occurs in a synchronous call the value of \$0 will be negative, the contents of ReturnCode will reiterate this value, ReturnText will describe the error, and ReturnBlob will be empty.

In this case the following test is sufficient:

```
ON ERR CALL("YourWebServiceErrorHandler")
CALL WEB SERVICE($SOAPServerAddress; $SoapAction; $Method; $NameSpace;
Web Service Dynamic)
```

```
If (OK=0)
```

```
    `Call failed or Server was not located. Handle your error here.
```

```
Else
```

```
    GET WEB SERVICE RESULT( $0; "ErrorCode")
```

```
    GET WEB SERVICE RESULT(vWSReturnText; "ReturnText")
```

```

$ReturnErr := $0
If($ReturnErr = 0)

Else
    ALERT("Call failed, error code = "+string($ReturnErr) + " " : " + vWSReturnText)
End if
End If
ON ERR CALL("")

```

The Asynchronous Case

All actions that include the strings “_NEW” or “_MODIFY” are interpreted by the server as being requests to update the database and are executed asynchronously in a separate process on the server. This does not apply to requests to delete data, which are associated with any action command that contains the string “_DELETE”. Deletions are handled synchronously.

After making a call to run one of these asynchronous actions the value returned in \$0 is ID number of the Batch Task record used to track the action. This will be a number greater than zero. If the number returned is equal or less than zero there has been an error and the action has been rejected. A description of the error will be returned in the variable named ReturnText.

When monitoring the progress of an asynchronous task you are concerned with two conditions rather than the one condition, that of success or failure, that you’re concerned with in the case of a synchronous task. In the asynchronous case you are also concerned with whether or not the task has finished. That’s because you only know that the task has completed successfully when there has been no error **AND** the task is finished. That is to say, until the task is finished the lack of any error is inconclusive.

The 4D code to extract and evaluate this condition is given in the following section.

The Batch_Status Action

Determine the current status of the ongoing action by issuing by making another Soap call using the Batch_Status action parameter and, at the same time, passing the ID number of the Batch Task in the DataBlob variable. This value needs to be assigned to the first element of a 1-element text array which is packed into the DataBlob variable using the Variable to Blob command. The format of this call is:

```

`Calling for an update on the status of an ongoing action.
Blob Size(vDataBlob;0)
Array Text(vTextArray;1)
vTextArray{1}:=string($BatchID_FromPreviousCall)
vBatchActin:="BATCH_STATUS"
Variable to Blob(vTextArray; vDataBlob)
SET WEB SERVICE PARAMTER("BatchAction"; vBatchAction)
SET WEB SERVICE PARAMTER("DataBlob"; vDataBlob)
CALL WEB SERVICE(vWEBTarget; vSoapAction; vMethod; v4QNameSpace; Web
Service Dynamic)

```

If this call is correctly formatted it will return zero in the \$0 parameter indicating that it was understood and the requested status information has been placed in a text array packed in the ReturnBlob variable. Extract this array using the Blob to Variable command.

As long as no error has occurred you only need be concerned with the content of the 0th and 2nd elements of this array. These elements tell you if the task has finished and if it succeeded. Should an error have occurred you'll need to refer to other elements in the array for a complete description. The full contents of this array are as follows.

Status Array Returned in ReturnBlob		
Element	Identifier	Value
0	"ErrorCode/"	Most recent error code value encountered in execution.
1	"Task_ID/"	Batch task ID that identifies this record.
2	"ActiveCode/"	activity value: +1 if still active, -1 if action completed.
3	"ref_FileNum/"	Target file number if this action involve updating data in this file.
4	"ref_RecID/"	Target record ID number if this action involve updating this record.
5	"Title/"	Title assigned to the task, generally a shortened version of the Action in {9}.
6	"ErrorCode/"	Most recent error code value encountered in execution
7	"TaskMessage/"	Concatenation of all update messages written to the task.
8	"Attention/"	Text tag manually assigned by the Administrator.
9	"Action/"	Action command that this task is executing.
10	"NumberAttempts/"	The number of current attempts to complete the action.
11	"Data (size)/"	Byte size of the DataBlob originally supplied, the Blob is not returned.
12	"DateSubmitted/"	Date the task was first submitted.
13	"TimeSubmitted/"	Time the task was first submitted.
14	"DateProcessed/"	Date the task action was concluded.
15	"TimeProcessed/"	Time the task action was concluded.
16	"User/"	Name of the user who submitted the task, for Soap actions this will always be the name of the server's proxy method _BTProxy4QSubmitTask.
17	"ReturnCode/"	Error code pertinent to Soap caller.
18	"ReturnText/"	Description of error pertinent to Soap caller.
19	"ReturnBLOB (size)/"	Byte size of the Blob returned, the Blob is not returned.

Each array element begins with the name of the data field it reports, a slash, and then the field's value. For example, the status call for ID = 100 would return the following value in this arrays first element:

Return Array Element{1} = "Task_ID/100"

To extract each field's value you must locate the slash and read the characters that follow it. For example to read the error and status values execute the following:

```
$SlashPos := Position("/", vTextArray{0})
$errorCode := Num(Substring(vTextArray{0}; $SlashPos + 1))
```

```

$NoErrors := ($ErrorCode = 0)
$SlashPos := Position("/"; vTextArray{2})
$ActivityCode := Num(Substring(vTextArray{2}; $SlashPos + 1))
$TaskComplete := ($ActivityCode = -1)

```

There are three conditions to be recognized in considering the status condition:

- 1 - Still Processing (activity code = +1)
- 2 - Finished processing, processing unsuccessful (activity code = -1, error code < 0)
- 3 - Finished processing, processing successful (activity code = -1, error code = 0)

Wait for the process to finish by entering a loop that periodically queries the server for the status of the asynchronously running task. Exit this loop when the task completes and take the necessary action.

The following code block assumes you have the correct syntax to delay the current process, to call the web service for the batch status, and also that an On Err Call method has been installed that assigns a true value to the vSoapError variable when a connection error occurs.

```

$ContinueWaiting:=True
While ($ContinueWaiting)
  Delay Process(...for 10 to 30 ticks...)
  CALL WEB SERVICE(...for batch status...)
  If (Not(vSoapError))
    GET WEB SERVICE RESULT(vWSReturnBlob; "ReturnBlob")
    If (Not(vSoapError))
      Blob to Variable(vWSReturnBlob; vTextArray)
      $SlashPos := Position("/"; vTextArray{2})
      $ActivityCode := Num(Substring(vTextArray{2}; $SlashPos + 1))
      $TaskComplete := ($ActivityCode = -1)
    End if
  End if
  $ContinueWaiting:= (Not($TaskComplete) & Not(vSoapError))
End While
$SlashPos := Position("/"; vTextArray{0})
$ErrorCode := Num(Substring(vTextArray{0}; $SlashPos + 1))
$NoErrors := ($ErrorCode = 0)
Case of
: (vSoapError) `alert user of communication error
: ($NoErrors) ` continue normal processing
Else
  $SlashPos := Position("/"; vTextArray{0})
  $ErrorMsg := Substring(vTextArray{18}; $SlashPos + 1)
  Alert ($ErrorMsg)
End Case

```

The necessary action is usually to continue with normal processing if the task completed successfully, and to alert the user, and take other appropriate actions, if the task completed unsuccessfully. These other actions could be to provide new data and resubmit the task, or to record the action as incomplete

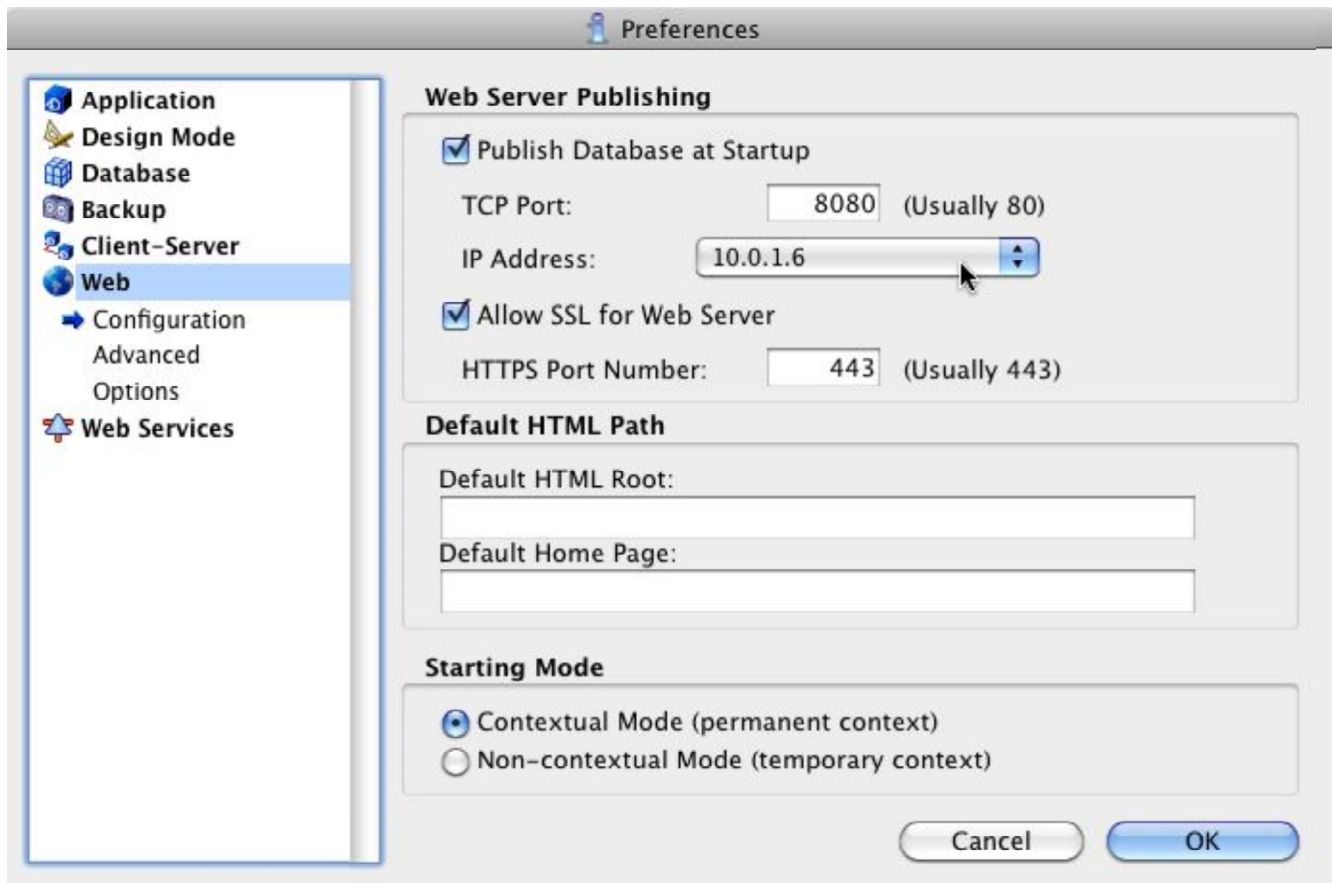
and give the user the option to return to the task at a later time.

Server-Side Issues

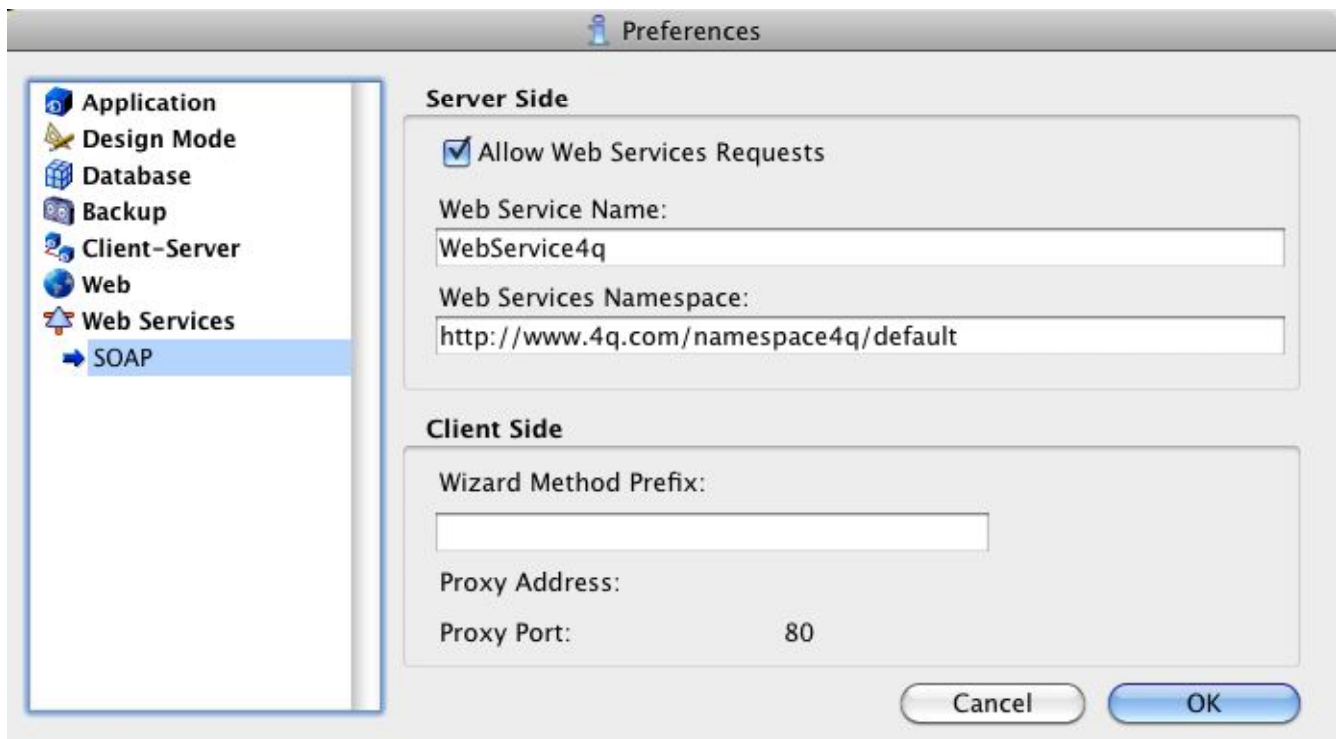
Starting Web Services

Start Web Services in a development or server version of 4D by checking the “Publish Database at Startup” checkbox in the Web Configuration pane of the 4D Preferences dialog, as shown below. In this pane set the TCP Port and IP Address. You will also need to check the “Allow Web Services Requests” that appears in the SOAP pane, also shown below.

I have found port 80 to be problematic on a Mac-based server platform and suggest port 8080, but any available port can be used as long as this specification is matched in the target URL used on the client side.



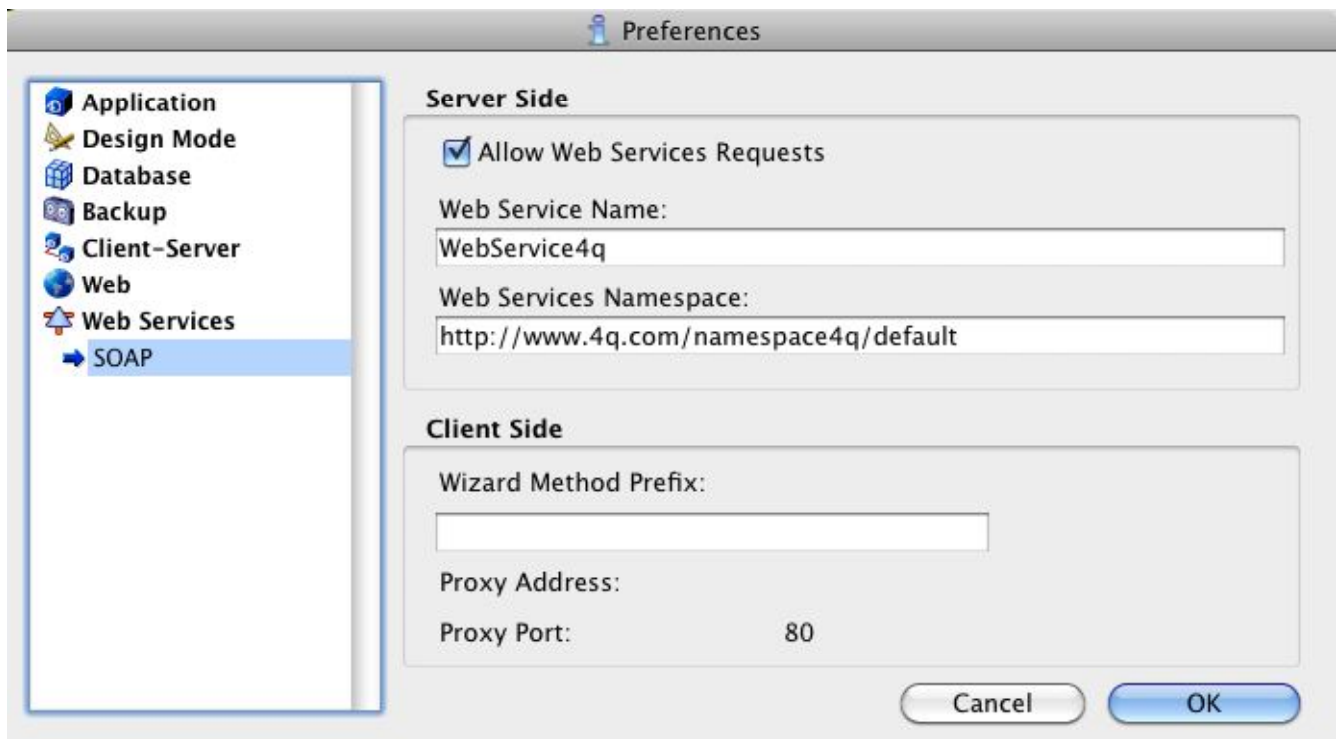
Make sure the “Allow Web Service Requests” is checked in the Server Side area of the SOAP preference pane. Also confirm that the Web Service Name is “WebService4q” and the Namespace is <http://www.4q.com/namespace4q/default>.



If you don't have a Web Service license issued by 4D, then your application will still act as a web server for 1 hour from the time that it's launched. Quitting and relaunching the application will restart the web server which will run for another hour. If your Soap client issues a Soap request after this 1-hour period it will receive a message indicating that the trial period has expired.

The Timeout Issue

For development purposes it would be useful to set the Timeout period to a larger number than would be acceptable in an installed application. 30 seconds might be appropriate for development whereas 10 seconds might be better for an installed application.



The problem with timing out is that if it occurs, the client may be in the dark as to what has or is happening on the server. Setting a low timeout limit does make sense when a data retrieval operation has stalled. In this case the server's response time should be short and whether or not the call times out there is no change in the data stored on the server.

In the case of a request that updates the server's data timing out could be problematic. Certainly the client must be allowed to continue execution without undue delay, but terminating the connection to the server is the worst way to do this. This highlights the flaw in using a synchronous protocol for operations that update data.

The 4Q Soap API attempts to resolve this issue by instituting synchronous communications only for operations that do not modify 4Q data, and instituting asynchronous communications for all operations that do.

This means that there should be no reason for a delay in server response in either case, even when there is a delay in performing a data updating process. For this reason it should be safe to assign a short timeout period of 3 to 5 seconds in the 4D Preferences pane. No 4th Quarter Soap request should require more than 3-5 seconds for a response. Those requests that involve the update of data on the server will be some of the quickest actions in generating a response, taking well less than 1 second in every case.

In summary, if a 4Q Soap call is taking more than 3 to 5 seconds then there is a communication problem and, most likely, the connection can be broken without leaving the client in any uncertainty about the integrity of the data on the server.

However, this is not to say that the client will immediately know the result of the action they have requested. In the case of data update actions the client only gets an ID in response to their request, a "ticket" if you will, that enables them to make subsequent inquiries as to the state of the action they

have submitted. In this case the completion of the action may be delayed for any amount of time. In the event that there is a intolerably long delay the client-side programmer will need to handle the situation themselves. The options available to the client have been discussed in the previous section titled “When to Monitor the Server: the Synchronous Case”.

Errors on the Server

You should be aware that when running in interpreted mode any error occurs in the proxy process spawned to handle the Soap request that this process immediately aborts. It does not halt, issue any message on the server, or drop the server into the debugger even when executing in interpreted mode, such as it would in any user-generated process.

This means that in order to debug errors on the server one must trace execution line by line until one reaches the command that causes the whole process to disappear. No message is generated when this line is run, the process simply crashes and the client, if the connection is still intact, will receive one of the cryptic error messages shown in the previous section titled “On Err Call Method.”

Also, any call to open a user interface object from within the Soap process will automatically crash the process. This includes opening a dialog or issuing the Alert. This applies only to the server, of course, and not to any code on the client.

Summary

Soap Client Sample Code

4Q Soap calls are handled either synchronously or asynchronously and each type of call is handled differently. Both types return a longint value in \$0 whose value is less than zero if there has been an error, zero if the call has completed successfully, and greater than zero if it refers to the ID of the Batch Task record that tracks an asynchronous operation.

Using the value returned in \$0 we can write a single client-side method for handling both types of calls without knowing whether which type of handling will be done on the server. The following 4D code combines the code segments quoted earlier in this document.

```
`General Format for 4Q Soap Client call method for synch- and asynch-type calls.
```

```
C_TEXT( vWSBatchAction; vWSReturnText; vWSReturnCode)
```

```
C_LONGINT( $0; $ReturnErr)
```

```
C_BLOB(vWSReturnBlob; vWSDataBlob)
```

```
ARRAY TEXT(vTextArray; 0)
```

```
SET BLOB SIZE(vWSReturnBlob; 0)
```

```
SET BLOB SIZE(vWSDataBlob; 0)
```

```
vSoapError:= False
```

```
$ReturnErr := 0
```

```
vWSBatchAction:= <some action value>
```

```
`If data needs to be passed in the Blob, assign values to text array(s) and
```

`then pack the array(s) into the DataBlob

ARRAY TEXT(vTextArray; <appropriate size>)

vTextArray {1} := `<assign values for action being undertaken>

vTextArray {2} := ...

...

VARIABLE TO BLOB(vTextArray; vWSDataBlob)

SET WEB SERVICE PARAMETER("BatchAction"; vWSBatchAction)

SET WEB SERVICE PARAMETER("DataBlob"; vWSDataBlob)

\$SOAPServerAddress := "http://10.0.1.6:8080/4DSOAP" `use correct IP address here

\$SoapAction := "WebService4q#_BTProxy4QSubmitTask"

\$Method := "_BTProxy4QSubmitTask"

\$NameSpace := "http://www.4q.com/namespace4q/default"

ON ERR CALL(<YourErrorHandler>) `Install your Web Service error handling method.

CALL WEB SERVICE(\$SOAPServerAddress; \$SoapAction; \$Method; \$NameSpace;

Web Service Dynamic)

If (vSoapError)

Call failed or Server was not located.

Else

GET WEB SERVICE RESULT(\$0; "ErrCode")

If (Not(vSoapError))

\$ReturnErr:= \$0

GET WEB SERVICE RESULT(vWSReturnCode; "ReturnCode")

GET WEB SERVICE RESULT(vWSReturnText; "ReturnText")

GET WEB SERVICE RESULT(vWSReturnBlob; "ReturnBlob")

Case of

: (\$ReturnErr <0)

Alert (vWSReturnText) `Action failed, deal with it.

: (\$ReturnErr =0)

Synchronous action completed successfully.

Else `Action is asynchronous, monitor its progress

vWSBatchAction:= "BATCH_STATUS"

\$BatchID := \$ReturnErr

ARRAY TEXT(vTextArray; 1>)

vTextArray {1} := **String**(\$BatchID)

VARIABLE TO BLOB(vTextArray; vWSDataBlob)

SET WEB SERVICE PARAMETER("BatchAction"; vWSBatchAction)

SET WEB SERVICE PARAMETER("DataBlob"; vWSDataBlob)

\$ContinueWaiting:=True

\$MaxWait:=15 `This is a Fail-Safe in case 4Q fails to respond properly

\$StartTime:=**Current Time**

While (\$ContinueWaiting)

Delay Process(**Current Process**; <...for 10 to 30 ticks...>)

CALL WEB SERVICE(\$SOAPServerAddress; \$SoapAction; \$Method;

\$NameSpace; Web Service Dynamic)

```

If (Not(vSoapError))
  GET WEB SERVICE RESULT(vWSReturnBlob; "ReturnBlob")
  If (Not(vSoapError))
    Blob to Variable(vWSReturnBlob; vTextArray)
    $SlashPos := Position("/",vTextArray {2})
    $ActivityCode := Num(Substring(vTextArray {2}; $SlashPos + 1))
    $TaskComplete := ($ActivityCode = -1)
  Else
    `Handle Soap communication error.
  End if
Else
  `Handle Soap communication error.
End if
$TakingTooLong:= (Current Time – $StartTime > $MaxWait)
$ContinueWaiting:= (Not($TaskComplete) & Not(vSoapError) &
  Not($TakingTooLong))
End While
$SlashPos:= Position("/",vTextArray {0})
$ErrorCode := Num(Substring(vTextArray {0}; $SlashPos + 1))
$NoErrors := ($ErrorCode = 0)
$SlashPos := Position("/",vTextArray {0})
$CurrentStatusMsg := Substring(vTextArray{7}; $SlashPos + 1)
$SlashPos := Position("/",vTextArray {0})
$TaskMsg:= Substring(vTextArray{18}; $SlashPos + 1)

Case of
: (vSoapError) `alert user of communication error
: ($TakingTooLong)
  Alert ("Connection to 4Q timed out - "+ $CurrentStatusMsg)
: ($NoErrors) ` continue normal processing
  Alert ($TaskMsg)      ` confirm success, this is just FYI, you would not
                        ` really bother the user with this confirmation
Else
  Alert ($TaskMsg)      ` inform the user of the error
End Case

End Case
Else
  `Handle Soap communication error.
End if
End If
ON ERR CALL("")

```