Compiled Xbase Pages (CXP/)

Xbase++ 2.0 has greatly expanded it internet capabilities and has introduced some exciting new technologies as well as improving on older technologies. This session will introduce users to techniques which will make it easier than ever to open your existing application data and reports to users via the web. Compiled Xbase++ Pages (<CXP/>) is a build and execution infrastructure that allows programmers to create powerful dynamic web sites.

Xbase++ 2.0 utilizes a new Socket layer, WebHandler() and HttpEndPoint() classes to provide the functionality for the creation of web servers and HTTP connections.

This session introduces the Xbase++ programmer to the following concepts:

* An introduction to the basics of the web server and web client (browser) relationship.

* What are compiled pages (<CXP/>/) ?

* How to set up a server to use compiled pages. Installing and configuring APACHE or IIS for <CXP/>.

* Setting up an Xbase++ environment for use with <CXP/>.

* Sample <CXP/> programs that deliver dynamic HTML and Javascript.

* How to use your own function libraries with <CXP/>.

* The basics of <CXP/> programming.

* How <CXP/> compares to scripting languages such as PHP or ASP.

* <CXP/> error handling.

* <CXP/> code debugging.

* Creating a web server using HttpEndPoint() and WebHandler().

* How to create your own <CXP/> web server using HttpEndPoint().

* How to create a web client/server application using WebSocketClient(), WebSocketHandler() and HttpEndPoint().

Web Server and Web Client Relationship

The primary function of a web server is to store, process and deliver web pages to clients (the web browser). The communication between client and server takes place using the Hypertext Transfer Protocol (HTTP). Pages delivered are most frequently HTML documents, which may include images, style sheets and scripts in addition to text content.

A user agent, commonly a web browser, initiates communication by making a request for a specific resource using HTTP and the server responds with the content of that resource or an error message if unable to do so. The resource is typically a real file on the server, but this is not necessarily the case and depends on how the web server is implemented.

While the primary function is to serve content, a full implementation of HTTP also includes ways of receiving content from clients. This feature is used for submitting web forms, including uploading of files.

Many generic web servers also support server-side scripting using Active Server Pages (ASP), PHP, <CXP/> or other scripting languages. This means that the behaviour of the web server can be scripted in separate files, while the actual server software remains unchanged. Usually, this function is used to generate HTML documents dynamically ("on-the-fly") as opposed to returning static documents. The former is primarily used for retrieving and/or modifying information from databases. The latter is typically much faster and more easily cached but cannot deliver dynamic content.

Here is a simple example. TEST.HTM contains the following content:

```
<!-- TEST.HTM -->

<body>
<H2>Hello from my Test Server</H2>
<IMG SRC="./images/donnay-logo.jpg" width=200>
</body>
```

The web browser requests TEST.HTM by connecting to the server via its IP address and port.

```
http://localhost:8080/test.htm
```

The contents of the file TEST.HTM are then sent to the browser. The browser must then interpret the HTML and render it. As the browser traverses the HTML content, it will encounter resources that exist on the server, such as the IMG tag that references ./images/donnay-logo.jpg. The browser will request that file using the same connection as the original connection and the server will transfer that file. This will continue until all resources such as JPG files, GIF files, etc. are retrieved by the browser and rendered within the displayed page.

These kind of files that are being transferred are referred to as "static documents".

What are Compiled Pages?

Compiled pages, hereinafter referred to as <CXP/>, contain combinations of Xbase++ code and HTML that return content to the web browser. The content must always be a string of data that is understood by the web browser. Web browsers understand HTML and Javascript, therefore the code that is executed when a <CXP/> page is loaded must always return such content. <CXP/> utilizes a technique known as FastCGI to accomplish this task very quickly. The two most common web servers that support FastCGI are IIS (Internet Information Services) and Apache. <CXP/> works very well with both of these web servers. A .CXP file would be requested instead of a .HTM file when the application needs "dynamic" content, such as information from a database. The .CXP page can also read information that is sent back to the server from a browser, such as a form of data or other data that is passed as a "query string".

.CXP pages are compiled similarly to the way .PRG code is compiled, except the <CXP/> language is much more flexible in its ability to embed the output of executed code within HTML and vice-versa.

Here is a simple example. TEST.CXP contains the following content:

```
<!-- TEST.CXP -->

<body>
<H2>Hello from my Test Server</H2>
The time on the server is @(Time()) and the date is @(Date()) <br>
<IMG SRC="./images/donnay-logo.jpg" width=200>
</body>
```

The web browser requests TEST.CXP by connecting to the server via its IP address and port.

```
http://localhost:8080/test.cxp
```

The contents of the file TEST.CXP are NOT sent to the browser. Instead, the web server calls a program named **cxp-worker.exe** and passes the name of the <CXP/> file to load. This relationship is previously set up when configuring IIS or Apache for <CXP/> during Xbase++ installation. Cxp-worker.exe will compile TEXT.CXP and create TEST.CXP.DLL in a sub-folder named **cxp-application**. The code in the DLL will then be executed and will be responsible for returning dynamic HTML to the web browser.

Web servers always work in a **stateless** environment. The purpose of a web server is to serve files or content to the web browser client when it is requested by the client. <CXP/> is a dynamic environment with the purpose of returning text such as HTML, XML, Javascript, etc. back to the web browser. Every time it does that it must load a lot of DLLs and then unload them before sending the content back to the web server via the CGI gateway. To work properly, all DLLs must be dynamically loadable and unloadable.

Why use <CXP/> for Web Development?

Most Xbase++ developers have applications which have been deployed for many years and are often asked by customers if they can access data or reports from a web browser at home. Some developers are already doing this (and much more) by using a third-party library named Xb2.Net. Xb2.Net allows the Xbase++ developer to create his/her own web server which resides on the same server as the customer data and runtime binaries. This concept has its drawbacks and limitations.

Every time the program needs to be updated, the executable must be terminated on the server, the binaries need to be rebuilt, transferred to the server and then restarted on the server. This usually means testing everything on a local machine and then deploying later to the server. This is not unusual or undesirable if it is a large application that requires more rigorous testing. It can be a grueling and laborious process however if only a small modification is needed or if a new application is needed. Imagine going through this process while making dozens of changes and then testing each change via the web browser.

When using <CXP/> pages for the application code, only the .CXP file(s) need to be updated and then deployed to the server. Programmers who have been developing web applications that use static .HTM files are keenly aware of such advantages. They know that that don't need to stop the web server, recompile it and restart it just to deploy a quick change. They know that they only need to send the .HTM file to the server and then reload the page on the browser. This is also true for <CXP/> pages. Furthermore, by using a product like WebDrive to map a server drive to a local computer, the developer can edit the HTM or CXP files without even requiring a FTP or RDP connection to the server.

When a <CXP/> page is updated, cxp-worker.exe (on the server) compares the date/time of the CXP file against the last DLL that was created from that CXP source. If the file was updated, the DLL is unloaded, recompiled and reloaded before calling the page. A <CXP/> application can be developed and tested on a local server and then deployed on a remote server by simply copying the updated .CXP files. If however, it is desired that no source code exist on the remote server, you can deploy only the *.cxp.dll files. This can be problematic because there must be a .CXP file on the server that has the same name as the associated .DLL (less the .DLL extension) or the program will not load, therefore you cannot simply copy up the *.cxp.dll files to the server. What you can do, however, is give the CXP source files an earlier date than the .dll files, then the CXP runtime will not try to recompile the .Dlls. You can create a dummy .CXP file of an earlier date and make sure that the file is empty.

My interest in <CXP/> was due to the fact that I maintain 4 different web sites n my Apache server. Each is in its own sub folder under C:\wamp\www. I found that it took too much time to keep these websites updated with static information. I needed it to be more dynamic. For example, the USS James E. Kyes website (http://ussjek.org) supports association members who were aboard this U.S. Navy destroyer from 1946-1973. The website tries to list all shipmates who were aboard during those years. This list needs to be constantly maintained and updated on the website. The roster is in a Microsoft Excel file and it is then saved as a set of .HTM files which are loaded by the website. I wanted to give shipmates the ability to see the list in 5 different orders, ie State, Years On board, Rank/Rate, etc. This required saving 5 different sets of HTM files (one for each sort order). I found myself constantly being reminded that the roster on the website is not up to date. <CXP/> made it possible to make this completely dynamic and update the list directly from a database which resides on the server. Now, I no longer use Excel and publish the updated HTML, instead I just make changes to the database.

Here is the <CXP/> program that I created in only about 1/2 hour from a template (**databrowse_template.cxp**) I created for the purpose of browsing databases.

```
<!--- Roster.cxp --->

<html>
 <head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
   <link rel="stylesheet" type="text/css" href="donnay.css" />
 </head>

<body>

<%

IF PCount() = 0 .OR. Empty(::session:rows)

  nRows := 100

  aCols := { ;
   {'Name',300,'Name',{||ROSTER->name}}, ;
   {'Address',500,nil,{||ROSTER->address}}, ;
   {'City',150,nil,{||ROSTER->city}}, ;
   {'State',150,nil,{||ROSTER->state}}, ;
   {'Zip',50,'Zip',{||ROSTER->zip}}, ;
   {'Phone',200,'Phone',{||ROSTER->phone}}, ;
   {'E-Mail',50,nil,{||ROSTER->email}}, ;
   {'Rate/Rank',50,'Rate',{||ROSTER->rate_grade}}, ;
   {'Record',50,nil,{||ROSTER->(RecNo())}}}

  cDatabase := 'ROSTER'
  cDbe := 'FOXCDX'
  cOrder := ''
  nRecNo := 0

  ::session:rows := nRows
  ::session:cols := aCols
  ::session:database := cDatabase
  ::session:dbe := cDbe

ELSE

  nRows := ::session:rows
  aCols := ::session:cols
  cDatabase := ::session:database
  cDbe := ::session:dbe
  cOrder := ::session:order
  nRecNo := ::session:recNo

ENDIF

oForm := ::httpRequest:form

aTDoption := Array(Len(aCols))
aTHoption := Array(Len(aCols))
AFill(aTDoption,'')
AFill(aTHoption,'')

cPath := ::physicalPath

USE (cPath + '\' + cDatabase) INDEX (cPath + '\' + cDatabase) VIA (cDbe) NEW
cAlias := Alias()

cAction := oForm:navigate
cSetOrder := oForm:setorder
cSeekValue := oForm:seekValue

cSeekMessage := ''

BEGIN SEQUENCE
IF Empty(cOrder) .AND. Empty(cSetOrder)
  (cAlias)->(OrdSetFocus(0))
ELSEIF !Empty(cSetOrder)
  IF cSetOrder = 'Record'
    cSetorder := 0
  ENDIF
  (cAlias)->(OrdSetFocus(cSetOrder))
  BREAK
ELSEIF !Empty(cOrder)
  (cAlias)->(OrdSetFocus(cOrder))
ENDIF

IF !Empty(cSeekValue)
  IF !(cAlias)->(dbSeek(Upper(Trim(cSeekValue))))
    dbGoTo(nRecNo)
    cSeekMessage := Upper(Trim(cSeekValue)) + ' not found!'
  ENDIF
ELSEIF nRecNo > 0
  (cAlias)->(dbGoTo(nRecNo))
ENDIF

END SEQUENCE

FOR i := 1 TO Len(aCols)
  aTHOption[i] := 'width=' + Alltrim(Str(aCols[i,2]))
  IF !(aCols[i,3]==nil) .AND. (cAlias)->(OrdSetFocus()) == Upper(aCols[i,3])
    aTHOption[i] += ' style="background-color:darkred"'
    aTDOption[i] := ' style="color:darkred"'
  ENDIF
NEXT

nRecNo := RecNo()
nRow := 1
%>

<form method='POST' action= './roster.cxp?submit'>

 Seek Value
 <input name='seekValue' value="@(Trim(Upper(cSeekValue)))" size=20>
 <input name='seek' value='Seek' type='Submit'>
 <b style="color:red">@(cSeekMessage)</b>

 <table border=0 class='dctable'>

   <tr>

     @FOR i := 1 TO Len(aCols)
       <th @(aTHOption[i])>
       @IF !(aCols[i,3])==nil
         <input name='setorder' type='submit' value=@(aCols[i,3]) style="font-size:20px">
       @ELSE
         @(aCols[i,1])
       @ENDIF
      </th>
     @NEXT

   </tr>

   <%
   DO WHILE (cAlias)->(!Eof())
   %>
     <tr @(IIF(nRow%2 == 0, 'style="background-color:lightyellow"','')) >
       <% FOR i := 1 TO Len(aCols) %>
         <% cData := Eval(aCols[i,4]) %>
         <td @(aTDOption[i])> @(cData) </td>
       <% NEXT %>
     </tr>
   <%
    nRow++
    (cAlias)->(dbSkip())

  ENDDO

  ::session:recNo := nRecNo
  ::session:order := (cAlias)->(OrdSetFocus())

  %>
 </table>

</form>

@((cAlias)->(dbCloseArea()))

</body>

</html>
```

If you already have a web solution that uses Xb2.Net or IIS or Apache, you can continue with that solution and yet still add functionality with <CXP/> at any time. <CXP/> pages run in their own environment and therefore do not require rebuilding any existing environment.

Understanding the <CXP/> Runtime Environment

<CXP/> is designed to encapsulate or localize an entire runtime environment so as to insure that applications always run smoothly in the event that the greater environment has changed. <CXP/> applications are installed in a sub-directory under the web server's ROOT folder. In the case of IIS, this is c:\inetpub\wwwroot. In the case of Wamp Apache, this is c:\wamp\www. Every other file needed by the application will usually reside in a sub folder or a sub-sub folder. The Xbase++ runtime will always reside in the same place, usually c:\Program Files (x86)\Alaska Software\cxp20. It is not always practical, however, to encapsulate an entire environment in a sub folder. For example, the application may require access to databases that are on a different drive or even a different server. What should be localized is the source code (.CXP, .HTM, .PRG, .CH, .JPG) files, and binaries (.LIB, .DLL) files that are compiled and called by the application.

Installing <CXP/> with Internet Information Server (IIS) 7 or higher

Before you begin you must establish whether you are using a desktop or a server operating system. Vista, Windows 7 and later versions belong to the desktop operating systems group. Windows 2008 and later are server operating systems. The installation procedure differs sligthly between these groups and is outlined in individual sections below. Further information on IIS configuration and administration can be found on http://www.iis.net. Once the IIS web server is properly installed, the installation package will properly
install the <CXP/> technology and will also apply proper configuration on the IIS web server for <CXP/> usage.

Vista and later

Open the Control Panel and select the item Programs. Under Programs and Features select Turn Windows features on or off. Then accomplish following steps. Please follow strictly the order below:

Ensure that the item Internet Information Service is selected. If this is not the case select the item Internet Information Service.

Also make sure that support for CGI executables is enabled which also enables the FastCGIModule. The CGI feature can be found at the following location in the feature tree:

Internet Information Services -> World Wide Web Services -> Application Development Features -> CGI

Windows 2008 and later

Open the Server Manager and scroll down to Roles Summary. Click Add Roles. Select the role Web Server (IIS) and click Next. Ensure that the role service CGI is selected which also enables the FastCGIModule. The role service CGI can be found on the following location:

Web Server -> Application Development -> CGI

Verify IIS/Web Server installation

To verify the IIS installation, open a web browser on your computer and navigate to http://localhost. The welcome page will be displayed if the web server was installed successfully.

To verify the <CXP/> installation, open a webrowse and navigate to http://localhost/cxpinfo.cxp.

HTTP Error 500.21 - Internal Server Error
Handler "CxpWorker" has a bad module "FastCgiModule" in its module list

If you get the above error when trying to load a .CXP page using IIS server, it is because you have not enabled the CGI role.

The below URL gives detailed instructions on how to do this.

http://www.iis.net/configreference/system.webserver/fastcgi

Installing <CXP/> with the Apache Web Server

These instructions are specifically for Wamp Apache (Apache version 2.2.22). This is the version of Apache that installs with Wamp Server. Wamp Server includes Apache, PHP and MySQL because this is one of the most popular installations for a variety of applications, including PhpBB3, a popular web forum application. However, with
minor modifications, these instructions apply to any installation of Apache. If you are using a version of Apache that is later than 2.2.22 then replace that version in the below instructions with the version you have installed.

If you follow these installation instructions step-by-step, you should get up and running very quickly.

1. Install Xbase++ 2.0 on a workstation or your server. It is not necessary to have the complete Xbase++ installation on your server. All that is needed on the server is the CXP20 folders.

2. After the Xbase++ 2.0 installation is complete, locate the following folder: C:\Program Files (x86)\Alaska Software\cxp20

3. If you already installed Xbase++ and <CXP/> on the same server or workstation as your Apache installation, then the environment will be ready for Apache. If you want to deploy only the <CXP/>

runtime on your Apache server then do the following (a) create a new folder on your server named C:\wamp\cxp20 (b) copy the entire C:\Program Files (x86)\Alaska Software\cxp20 folder, including sub-directories to the new C:\wamp\cxp20 folder on your server.

4. Locate the following folder on your workstation or server: **C:\inetpub\wwwroot**. If you installed <CXP/> when you installed Xbase++ 2.0, then the installation program will have copied some <CXP/> files to this folder. The file cxpinfo.cxp will be installed to c:\inetpub\wwwroot only if IIS is installed.
5. Locate the following folder on your server: C:\wamp\www. This is the root folder for Apache. It is defined in the httpd.conf file of Apache as the root folder with the DocumentRoot entry. If you not using WAMP, then it will be defined as a different location.

6a. Copy all *.CXP files in the C:\inetpub\wwwroot folder to the C:\wamp\www folder on your server. If there is a c:\inetpub\wwwroot\cxp-application folder it does NOT need to be copied to your server. This folder is automatically maintained by <CXP/>. That folder does not even exist if the cxpinfo.cxp page was not executed previously.

6b. Copy the c:\inetpub\wwwroot\websamples folder to the c:\wamp\www folder on your server. The websamples provide a source of information illustrating usage and filling gaps where the documentation is weak.

7. Download the following file from the Donnay Software website:
http://bb.donnay-software.com/download/mod_fcgid-2.3.6-win32-x86.zip **(32 bit)**
or
http://bb.donnay-software.com/download/mod_fcgid-2.3.6-win64.zip **(64 bit)**
then extract the mod_fcgid.so file into the C:\wamp\bin\apache\apache2.2.22\modules folder on the server. This is the module that handles FastCGI.

8. Run the C:\cxp20\bin\apachecfg.exe with the following parameters:
-i -c=C:\wamp\bin\apache\apache2.2.22 -n=c:\wamp\cxp20\bin\cxp-worker.exe. This will create a backup of your c:\wamp\bin\apache\apache2.2.22\conf\httpd.conf file and will add new entries that will enable the <CXP/> system and FastCGI.

Hint: apachecfg.exe -h prints the usage of the program.

9. Stop and restart your wampApache service. Note: this may not be necessary if Apache is already running because apachecfg.exe normally restarts the service. It will only be necessary if you had to make additional modifications to your httpd.conf file (which is not recommended).

10. Open a web browser on your server and navigate to http://localhost/cxpinfo.cxp. This should show your first <CXP/> page. After you confirm that everything is running ok, then navigate to http://localhost/websamples/index.cxp to run the sample selection.

NOTE: If you did not get Apache from the WAMP installation, then the Apache system will have a different installation folder on your server and may also be a different version. The above instructions are for WAMP Apache 2.22.22. Just modify the instructions to accommodate your directory structure.

After configuration, your Apache HTTPD.CONF file will have the following additions:

```
#CXP Start Installer automatically modified
#Do not manually modify here! This will be removed on deinstallation.
LoadModule fcgid_module modules/mod_fcgid.so
#CXP End Installer automatically modified

<Directory "c:/wamp/www/">
  #CXP Start Installer automatically modified
  #Do not manually modify here! This will be removed on deinstallation.
    <IfModule fcgid_module>
       <Files *.cxp>
         Options +execCGI
       </Files>
    </IfModule>
  #CXP End Installer automatically modified
</Directory>

#CXP Start Installer automatically modified
#Do not manually modify here! This will be removed on deinstallation.
<IfModule fcgid_module>
    # We allow only one request per instance thus we need more processes then default
   FcgidMaxProcesses      500

    # Process start and process end each increment the score. On each interval clear the score
    FcgidTimeScore         1000

    AddHandler fcgid-script .cxp
    FcgidWrapper    "c:\cxp20\bin\cxp-worker.exe" .cxp
    FcgidCmdOptions "c:\cxp20\bin\cxp-worker.exe" InitialEnv TEMP=C:\Windows\TEMP
MaxRequestsPerProcess 1
</IfModule>
#CXP End Installer automatically modified
```

Known anomolies with Apache

When calling the CXP page from the web browser, Apache is case sensitive.
This will NOT work: http://localhost/dd787.Cxp
This WILL work: http://localhost/dd787.cxp

The CXP page cannot be given the same name as the root directory.
This will cause an Internal Error: http://localhost/dd787/dd787.cxp
This will work: http://localhost/dd787/rsvp787.cxp

A CXP layout file cannot be given the same name as a CXP file.
This will cause an Internal Error

Using your own libraries with <CXP/>

<CXP/>, by default, is only aware of the Xbase++ runtime libraries (DLLs), the compile-time libraries (LIBs) and the compile-time includes (CHs). If your web application needs access to your own DLL, LIB and CH files, then you must make them available to cxp-worker.

Here are instructions for setting up <CXP/> to use your own libraries that have been compiled in Xbase++.

Under the WWW root directory of your <CXP/> application, you will need to create a new folder named Helpers. For example, below is the <CXP/> source code for an application that lists customers from a database. The application is in a file named CUSTOMER.CXP under C:\wamp\www\ds\customers (This example is for Apache). If it were using IIS then it would be in c:\inetpub\wwwroot\ds\customers.

In the below example, I created a folder named C:\wamp\www\ds\customers\Helpers and then copied DCLIPX.LIB and DCLIPX.DLL into the Helpers folder.

I then created a file named C:\wamp\www\ds\customers\application.config. It has the following content:

A sample application.config file:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>

<info author   = "Roger Donnay"
  version  = "1.0"
  copyright= "Copyright (C) $YEAR$ Donnay Software Designs. No rights deserved"
  title    = "Customers" header   = "Program for listing
  Sample Customer Database" />

<!-- List the libraries to be linked with the CXP page -->
<helpers
   lib="dclipx.lib"
/>
```

Unfortunately, CH files are a different matter. They must reside in the C:\Alaska Software\Xbase++\Cxp20\Include folder. This will be changed in a future <CXP/> update in which CH files can also be added to the Helpers.

A sample CUSTOMER.CXP file that uses your library

```
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
</head>

<body>
<h1>My Customer List</h1>
<h3>Rendering as simple text</h3>
<pre>
<%
   USE (::PhysicalPath+"customer.dbf") ;
     INDEX (::PhysicalPath+"\customer.cdx");
     VIA "FOXCDX"
   oRecord := CUSTOMER->(DC_DbRecord():new())
   DO WHILE !CUSTOMER->(Eof())
     CUSTOMER->(DC_DbScatter(oRecord))
     ? oRecord:bill_name, oRecord:bill_strt, oRecord:bill_city, oRecord:bill_state
     CUSTOMER->(dbSkip())
   ENDDO
 %>
</pre>
</body>
</html>
```

Caveat:

It is possible to create an Xbase++ library that is linked to Xbase++ runtime DLLs that are not included in the C:\Program Files (x86)\Alaska Software\cxp20\bin folder. This would cause a failure of a .CXP program to load if that library is used in your .CXP program. For example, I found that one of my .CXP programs would not load when using a .DLL that required the Xbase++ DLLs **ASUED10.dll** and **ASUED10C.dll**. These DLLs exist in the **C:\Program Files (x86)\Alaska Software\xpp20\runtime** folder and must be copied to the **C:\Program Files (x86)\Alaska Software\cxp20\bin** folder. A load error will not be obvious because there will be no CXP error other than something similar to the following displayed in the web browser: **\inetpub\wwwroot\medalion\cxp-application\drivers.cxp.dll.**

If you experience such a problem, then there is a simple method for determining the cause. Use the CHK4DLL.EXE utility to test your DLLs to make sure that they will load when your path is set to the CXP bin folder.

Another problem can be caused by trying to load a .DLL that is **not unloadable**. This is another issue that I ran into several times. For example, my DCLIPX.DLL is my main DLL for my eXpress++ product. I like to also use functions from this DLL in my CXP applications. Recently, I added some new functionality that uses the XbpHtmlWindow class. This class exists in the Xbase++ **xppwui.dll**. All of my <CXP/> programs suddenly stopped working when I updated my Xbase++ libraries to the most current version. This is because Alaska Software did something to xppwui.dll which made it not unloadable. To determine why my DCLIPX.DLL would no longer work with <CXP/> I needed to run the Xbase++ utility program: DLLINFO.EXE. When I did that, I discovered the problem:

```
DllInfo - DLL info utility Version 2.00.721
Copyright (c) Alaska Software 2000-2016. All rights reserved.

Path name    : C:\Program Files\Alaska Software\xpp20\lib\xppwui.dll
Type         : Xbase++ not prepared for dynamic unload
```

When using your own libraries with <CXP/> it is a good idea to check them often using DLLINFO.EXE to insure that they are <CXP/> compatible. I needed to create a custom version of my library that excluded any functions that were in xppwui.dll.

The basics of <CXP/> programming

The Language Reference

`<%...%>` code section

A <CXP/> file normally contains HTML tags, just like an HTML file. However, a <CXP/> file caan also contain code sections surrounded by the delimiters <% and %>. These code sections are executed on the server and can be made up of any expression, statement, procedure, or class valid in the Xbase++ language.

Inside a code section, the ? command is used to write to the current output section. The following example writes the string "Hello World" into the body element of the document.

```
<html>
<body>
<%
 ? "Hello World!"
%>
</body>
```

```
</html>
```

The command ? internally uses the method ::HttpResponse:WriteStr(), which performs HTML-encoding and appends a CRLF-pair to the string. Data without a line break can be output using the command ??, which uses the ::HttpResponse:Write() method instead. To output the data as-is, without any encoding or line breaks, the method ::HttpResponse:WriteRaw() must be called directly.

```
<html>
<head>
<!-- Output the version data inside the title -->
<!-- without a crlf pair                       -->
<title>Grettings from <%??Version()%></title>
<body>
<%

 // This iteration repeats the output, adding 16 lines
 // of text to the resulting page
 FOR n:=1 TO 16
   ? "<p style='background-color:#f5"+StrZero(n*6,2)+"20'>Hello world v2!</p>"
 NEXT n
%>
</body>
</html>
```

@(...) inline expression

@(<expression>)

The @(<expression>) syntax allows to embed expressions directly inside your declarative HTML markup in a simple and distinct way. The following sample code shows how the expression is transformed by the CxcBuilder to output the expression result in the current section.

```
<h2>Today is @(Date())!</h2>
```

Code:
```
::HttpResponse:WriteStr("<h2>Today is ",Date(),"!</h2>")
```

Note: Note that the output is automatically HTML-encoded! Therefore, inline expressions can not be used to write raw data to the output section.

@... single-line code

@<expression>|<command>|<statement>

When using the @ command, the entire line is declared as code. The line continuation character (;) can be used to break a single statement into multiple lines. The @ command allows writing code without the open/close markers normally required to identify code sections. In sections containing alternating lines of Xbase++ code and HTML markup, using the @ command increases readability and avoids unecessary noise. This is demonstrated in the code fragment below. Please note that it is not allowed to have multiple @ commands in the same line.

```
<%
x := 125
```

```
aDir := Directory()
%>

@IF x>10
  <h1>x is larger than 10</h1>
@ELSE
  <h1>y is smaller or equal to 10</h1>
@ENDIF

<h2>Directory Listing</h2>
<ul>
@FOR n:=1 TO Len(aDir)
  <li><pre>@(aDir[n,1]), @(aDir[n,2]), @(aDir[n,3]), @(aDir[n,4])</pre></li>
@NEXT n
</ul>
```

#code directive

Controls where code is emitted by the CxcBuilder

```
<%#code locality=<locality-name>%>
```

The #code directive is used to specify the locality of the code section following the directive. The supported locality tokens are described in the table below. The default locality is "page-render". The CxcBuilder automatically switches back to that locality after it has emitted the code section immediately following a #code directive.

Locality Tokens

Token	Description
*) Default locality	
page-init	Code section is injected at the end of the :Init() method of the page class
page-load	Code section is injected into the :Load() method of the page class. :Load() is executed when the page binary is loaded into the worker process.
page-unload	Code section is injected into the :Unload() method of the page class. :Unload() is executed when page binary is unloaded from worker prozess. This usually occurs before worker process gets recycled. The exaxt time is not determined.
page-render*)	Code sections is injected into the :Render() method of the page class. :Render() is where the HTML output (view content) is created.
page-global	Code section is injected into module header of the page class implementation file. Use this locality for defining your own functions, classes or static declarations. #include statements should also have be placed in this locality.

Note: page-load/page-unload is a good place for data base connection such as DacSession() connect / disconnect or table USE operations, as this will increase performance if worker process is used to handle multiple requests.

```
<%#code locality="page-global"%>
```

```
<%
// This section is injected into the module header
STATIC aData := { 3, 5, 7, 11 }

FUNCTION MyHelper(cInput)
RETURN(cOutput)
%>
<!-- this comment and the following code goes into -->
<!-- the page-render method.                       -->
<h1>
<%
 ? Version()
%>
</h1>
```

#page directive

Controls page behaviour

```
<%#page layout=<filename>
        trace="yes|no"
        implements=<classname>
        inherits=<baseclass> %>
```

Using the #page directive the behaviour of a page can be customized. The table below shows the available page token/value pairs.

page tokens

Token	Description
layout	Defines the layout master file for this page
trace	Enables/disables tracing output created via the :Trace() method for the debugger or the DebugView utility. The value must be specified as a string literal.
implements	Allows to change the name of the page class. By default, the class name is derived from the filename of the cxp page.
inherits	Allows to override the base class of the page class. By default, CxpAbstractPage is used.

#region directive

Controls display of code in the source code editor

```
<%#region <your text>%>
<%#end%>
```

The #region directive allows to specify a block of code that can be expanded or collapsed in the source code editor. By default, content embedded into a #region/#end directive is displayed collapsed and is not visible in the editor. Instead, only the text defined in <your text> is shown. This feature called "code folding" can contribute significantly to source code readability when using the Workbench.

Using Layout files, the RENDER command and the SECTION command

The RENDER command is used in layout files to specify the location in which a specific page section is rendered. If a layout is referenced in a page, page composition is defined in the layout file. In this case, the RENDER command must be used to selectively render the output defined in the content page. To do this, the name of the section must be specified.
By default output of a content page is written to the BODY section. Therefore at least the RENDER BODY command has to be used in the layout file. Otherwise the output of the content page is lost.

Sample layout file ("hello1.layout")

```
<html>
<head>
  <title>MySite by @(Version())</title>
</head>
<body>

  <div style="font-size:18pt;background-color:#ff0000;color:#efefef;">
  MyApplication
  </div>

  <!--- output from content page is rendered here -->
  @RENDER Body
  <div style="font-size:20pt;background-color:#0000ff;color:#efefef;">
    @RENDER Footer
  </div>

  @RENDER Hello

</body>
</html>
```

CXP Page with layout reference ("hello.cxp")

```
<%#page layout="./hello1.layout" %>

@SECTION Body
<h2>This text and the text in the footer are defined in page hello.cxp</h2>
<h3>The heading, colors and the overall page structure are defined in layout
file simplesite.layout</h3>

@SECTION Footer
<p>Nothing special today</p>

@SECTION Hello
<H2>Hello from my Test Server</H2>
The time on the server is @(Time()) and the date is @(Date()) <br>
<IMG SRC="./images/donnay-logo.jpg" width=200>
<p>
```

<CXP/> Error Handling

The <CXP/> infrastructure knows about different error types. Each error is demonstrated by the following CXP page:

```
// compile error
  SD SD
  FOR x:=1 TO 10
    ? "TaTa"
  ENDIF

// link error
  TaTa()

// runtime error
? test test
%>
```

When you run this code you will see the error information returned to the web browser with a detailed description of the error. Only the first error encountered will be returned, first compile errors, then link errors, then runtime errors.

<CXP/> Class Reference

As of the writing of this document, much of the class reference documentation has not been completed however perusing the .CXP files in c:\inetpub\wwwroot\websamples will show how these classes are used.

CxpAbstractPage()

This class implements the core features for any <CXP/> page created by the CxcBuilder.

All <CXP/> pages are by default derived from this class. Whenever a <CXP/> page needs to be rebuild, the CxcBuilder first creates an intermediate PRG file for the page. This file contains the implementation of a class which by default is derived from CxpAbstractPage. You can override this default behaviour by using the inherits directive.

Unless specified otherwise, the code and HTML markup contained in a CXP file is emitted into the :Render() method of the page class. This means that for a simple <CXP/> page, the framework just executes :Render() to display the page at runtime. In this case, the HTML markup together with any output generated with the ?/?? commands will be used to create the response send back to the browser after a page request.

For more complex scenarios, <CXP/> applications can use the code-locality directive to control the location where code is emitted in the intermediate PRG file created by the CxcBuilder. This way, application code can be made to execute not only during :Render(), but also when a page is loaded, for example. In addition, by using the HttpResponse Object and its dedicated output methods, the :Render() method of a <CXP/> page can return other content than just plain HTML.

Configuration

:CachePolicy	Determines the cache policy of the page.
:ConfigFile	Fully-qualified filename of the config file of the page.
:Config	Represents the root node of the configuration data for the page.

Properties

:Application	References the CxpApplication object the page belongs to.

:Session	References the session object handling session state for the page.
:Host	References an object for accessing functionality of the host machine or WWW server the <CXP/> application is running on.
:Data	References a container object which handles the data associated with the page.
:Params	Contains a container object for the parameters passed to the page.
:File	Contains the fully-qualified URL of the <CXP/> file, such as localhost/samples/index.cxp.
:Path	Contains the path to the page, such as localhost/samples/.
:PhysicalPath	Contains the path of the physical file location of the page, eg. c:\inetpub\samples\.
:HttpRequest	References a HttpRequest object representing the incoming request to the page.
:HttpResponse	References a HttpResponse object representing the response created by the page.
:Trace	Enables or disables tracing output.

Layout Management

:HasLayout()	Determine if the page has a layout associated.
:HasSection()	Determine if a specific Section has been defined
:RenderSection()	Renders the content of the given section by name
:SelectSection()	Selects a section by name.

Deferred Methods

:Render()	Renders the page content which is returned to the client browser.
:Load()	Gets executed when the page is loaded into memory.
:Unload()	Gets executed when the page is unloaded from memory. Page unloading occurs when a <CXP/> worker process is recylced or when the www server shuts down.

Error-handling and tracing-support

:Trace()	Output trace information to the Windows Debug interface.
:ErrorHandler()	The default handler for runtime errors occuring during page execution.

CxpApplication()

The function CxpApplication() returns the class object of the CxpApplication class.

The documentation is not completed.

HttpManager()

The class *HttpManager* controls the behaviour of the application on the HTTP protocol-level.

Configuration

:UseUploadFolder	Specifies whether uploaded files are stored in a local drive
:UploadFolder	The folder where uploaded files are stored in

HttpRequest()

Objects of this class encapsulate the incoming request from the client.

Properties

:Form	Provides access to the form variables of the request.
:Cookies	Provides access to the cookies of the request.
:Files	Provides access to the files uploaded from the browser.
:Headers	Provides access to the headers of the request.
:DocumentRoot	Contains the physical path to the document root of the site.
:QueryString	Contains the query string as posted by the browser.
:AcceptLanguages	Provides access to the accept languages order as posted by the browser.
:ContentEncoding	Contains the content encoding of the request.

Manipulation

:GetVariable()	Retrieve the value of a CGI/FastCGI variable.

HttpResponse()

Objects of this class encapsulate the outgoing HTTP response.

Properties

:ClientCachePolicy	Controls the browser's client cache policy for the response.
:ContentType	Contains the content type requested by the client.
:ServerCachePolicy	Determines the cache policy of this request.
:Response	The response as send to the server.

HTTP Header Management

:AddHeader()	Add a header token and value to the response.
:DelHeader()	Remove a header token from the response.
:DelHeaders()	Remove all headers, or just the headers with a given token from the response.
:GetHeader()	Get a header value given its token.

Cookie Management

:AddCookie()	Add a cookie.
:DelCookie()	Delete a cookie given its name.
:DelCookies()	Delete all cookies.

Manipulation

:WriteRaw()	Write an arbitrary value to the response.
:Write()	Write an arbitrary value to the response.
:WriteStr()	Write one or more values to the response with CRLF appended.
:WriteBinary()	Write binary data directly to the response.
:WriteFile()	Sends back a file to the client.
:Transfer()	Transfers the execution path to another page or controller.
:Redirect()	Redirects the browser to another URL.
:Reset()	Discards any pending content.
:Send()	Sends the response to the client.

HttpUploadedFile()

Instances of the class *HttpUploadedFile* represent files which are uploaded from within a <CXP/> page. The *HttpUploadedFile* objects are created automatically by the <CXP/> infrastructure and can be accessed via the :Files instance variable of the *HttpRequest* object.

Properties

:Data	Contains the file content.
:IsOnDisk	Specifies whether the file is stored on disk.
:LastError	Contains the error code of the last file operation.
:NameOnDisk	Contains the full pathname of the file on the web server's hard drive.
:Name	Contains the filename.
:Size	Specifies the size of the file.

<CXP/> Program Debugging

I have found that debugging a <CXP/> program can be very challenging when running under IIS or Apached. This is due to the fact that the <CXP/> runtime does not interact with the desktop. It is intended to work like any other server-based service. Therefore you must use great care to insure that nothing will stop program execution. For example, a call to MsgBox() will display nothing but will stop the program. You must then recycle IIS or Apache to unload the program. Anything that outputs to the screen will be ignored. This is one of the reasons I choose to do all my development and debugging using **CxpHttpServer.Exe,** discussed later in this session.

The <CXP/> Sample Programs

Alaska software has provided many sample programs to demonstrate the power and flexibility of <CXP/>. If you have installed for IIS, these sample programs will reside in c:\inetpub\www_root\websamples. If you have installed for Apache (WAMP version), these sample programs will reside in c:\wamp\www\websamples. To run the samples just point your web browser to http://<domainname>/websamples/index.cxp. It is also recommended that you run CXPINFO.CXP. This sample program demonstrates much of the information that is available to a <CXP/> application.

How to create your own Web Server with HttpEndPoint()

An object of the class *HttpEndpoint* serves as a connection point for incoming HTTP or WebSocket connections. The connection point is active after calling the method :start(). *:start()* automatically creates a new thread for listening on the port associated with the HttpEndPoint. The port number must be passed as a parameter to the method :new().

For HTTP requests with an URL referring to a local file, the HttpEndpoint loads the file from the hard drive and returns it to the connected web client. The following table lists the extensions and the corresponding MIME types of the resources which are supported by default:

Default extensions and MIME types

Extension	MIME type
.html	text/html
.htm	text/html
.gif	image/gif
.png	image/png
.jpg	image/jpeg jpeg jpg jpe
.js	application/x-javascript
.css	text/css
.woff	font/woff
.eot	application/vnd.ms-fontobject
.otf	application/octet-stream
.ttf	application/octet-stream
.svg	image/svg+xml
.json	application/json

A web request to a resource without an extension must be handled in application code. For this, a user-defined class must be written containing the logic for handling the request. The class must have the same name as the resource which is requested, and it must be derived from a base class which corresponds to the connection type.

If the client tries to establish a websocket connection, an object of a user-defined class that must be derived from *WebSocketHandler* is created. The following URL instantiates an object from the class MyWebSocketHandler:

```
ws://<servername>:<port>/MyWebSocketHandler
```

In case no websocket connection is to be established, an object is instantiated from a class that must be derived from the class *WebHandler*. The following URL instantiates an object from the class MyWebHandler, and calls the method :echo(). The parameters specified in the URL are forwarded to the handler method:

http://<servername>:<port>/MyWebHandler/echo?paramname1=value1

The following code fragment demonstrates the implementation of the handler that corresponds to the URL above:

```
CLASS MyWebHandler FROM WebHandler
  EXPORTED:
    METHOD echo
ENDCLASS

<...>

METHOD MyWebHandler:echo( cParamName1 )
   LOCAL cHtml
   cHtml := ""
   cHtml += "<html><title>MyWebHandler:echo() was called</title><body>"
   cHtml += "Parameter cParamName1 is " + cParamName1
   cHtml += "</body></html>"
RETURN cHtml
```

Class methods

:new()	Creates an instance of the class *HttpEndpoint*.

Connection

:start()	Start the listener thread
:stop()	Terminates the listener thread.

Implementing a web endpoint

```
// The example creates a html file in the current directory.
// The HttpEndpoint loads the file and returns it to the connected client on request.
#include "inkey.ch"
#include "web.ch"

#define PORT 81
PROCEDURE Main()
  LOCAL cPort
  LOCAL oHttpEndpoint
  LOCAL nKey
  LOCAL cHtml

  //
  // Creates a sample html file on the local drive.
  //
  IF .NOT. File( "default.html" )
     cHtml := ""
     cHtml += "<h1>Hello World!</h1>"
     cHtml += "<h2>Greetings from: "+AppName()+"</h2>"
     MemoWrit( "default.html", cHtml )
  ENDIF

  cPort := AllTrim(Str(PORT))
  ? "HttpEndpoint sample."
  ?
  ? "With a web browser navigate to:"
  ?
  ? "     http://localhost:"+cPort+"/default.html"
  ?
  ? "Press ESC to quit"

  //
  // Create an end point for handling incoming
  // connections on the specified port
  //
  // Note: Requests to .html files are handled
  //       automatically by the Http end point!
  //       No code must be written for returning
  //       the sample .html file.
  //
  oHttpEndpoint := HttpEndpoint():new( PORT )
  oHttpEndpoint:start()

  // Continue after the ESC key was pressed
  nKey := 0
  DO WHILE nKey <> K_ESC
     nKey := Inkey(1)
  ENDDO

  //
  // Stop the endpoint before the application
  // terminates
  //
  oHttpEndpoint:stop()

RETURN
```

WebHandler()

On web connections the class HttpEndpoint instantiates objects from user defined classes that must be derived from the class *WebHandler*. Those sub classes implement methods being called to serve the request.

Incoming

:execute()	Handle the request and provide the content to be returned to the client.

Properties

:HttpRequest	References a HttpRequest object representing the incoming request to the web handler.
:HttpResponse	References a HttpResponse object representing the response created by the web handler.

Implement a web endpoint

```
// The example shows a server that implements the Endpoint
// CurrentTime/get. Requests to other endpoints on CurrentTime
// encounter a default handling

#include "inkey.ch"
#include "web.ch"

#define PORT 81
PROCEDURE Main()
  LOCAL cPort
  LOCAL oHttpEndpoint
  LOCAL nKey

  cPort := AllTrim(Str(PORT))
  ? "Time Server sample."
  ?
  ? "With a web browser navigate to:"
  ?
  ? "     http://localhost:"+cPort+"/CurrentTime/get"
  ? "     http://localhost:"+cPort+"/CurrentTime/unknown"
  ?
  ? "Press ESC to quit"

  //
  // The listener thread is established
  //
  oHttpEndpoint := HttpEndpoint():new( PORT )
  oHttpEndpoint:start()

  // This sample is not doing something special
  nKey := 0
  DO WHILE nKey <> K_ESC
     nKey := Inkey(1)
  ENDDO

  //
  // Stop the endpoint before the application
  // is terminated
  //
  oHttpEndpoint:stop()

RETURN

//
// Handler for the current time
//
CLASS CurrentTime FROM WebHandler
  EXPORTED:
    METHOD get
    METHOD execute
ENDCLASS

//
// The method :get() handles the RESTful path CurrentTime/get
//
METHOD CurrentTime:get()
  LOCAL cThreadId
  LOCAL cResponse := ""

  cThreadId := AllTrim(Str(ThreadId()))

  cResponse += "<!DOCTYPE html>"
  cResponse += "<html>"
  cResponse +=   "<head><title>"+ProcName()+"</title></head>"
  cResponse +=   "<body>"
  cResponse +=     "<h1>Current Time:</h1>"
  cResponse +=     "<p>"+Time()+" (HH:MM:SS)</p>"
  cResponse +=     "<small>By thread: "+cThreadId+"</small>"
  cResponse +=   "</body>"
  cResponse += "</html>"

RETURN cResponse

//
// Dispatch the request to existing methods. Return a generic
// web page for all other requests.
//
METHOD CurrentTime:execute( cMethodName )
  LOCAL cResponse := ""

  IF IsMethod(SELF, cMethodName)
    RETURN SUPER:execute( cMethodName )
  ENDIF

  cResponse += "<!DOCTYPE html>"
  cResponse += "<html>"
  cResponse +=   "<head><title>Unhandled request</title></head>"
  cResponse +=   "<body>"
  cResponse +=     "<h1>No handler for: "+cMethodName+"</h1>"
  cResponse +=     "<p>Please try with: /CurrentTime/get</p>"
  cResponse +=     "<small>Sorry</small>"
  cResponse +=   "</body>"
  cResponse += "<html>"

RETURN cResponse
```

How to create a CXP Web Server with HttpEndPoint()

The below code will be supported in the near future by Alaska Software.

It is understood that the full capabilities of <CXP/> will be supported without the need for IIS or Apache. This is an example of how easy it will be to add <CXP/> support to your custom web server.

```
/// <summary>
/// We override DefaultProcessor so we can add our
/// CXP and HTMLFile processors which we need for
/// web debuging.
/// </summary>
CLASS CxpHttpEndPoint FROM HttpEndPoint
  EXPORTED:
  METHOD DefaultProcessors()
  METHOD DefaultConfig()
ENDCLASS

METHOD CxpHttpEndPoint:DefaultProcessors()
  LOCAL aRet := SUPER:DefaultProcessors()
  AAdd( aRet , "CxpProcessor" )
  AAdd( aRet , "HtmlFileProcessor" )
RETURN(aRet)

METHOD CxpHttpEndPoint:DefaultConfig(cToken)
  LOCAL oDO := DataObject():New()

  IF( !("CDXDBE" $ dbelist()))
    DbeLoad("CDXDBE")
  ENDIF
  IF( !("FOXDBE" $ dbelist()))
    DbeLoad("FOXDBE")
  ENDIF
  IF( !("FOXCDX" $ dbelist()))
    DbeBuild("FOXCDX","FOXDBE","CDXDBE")
  ENDIF

  cToken := Lower(AllTrim(cToken))

  IF(cToken=="session")
    oDO:Provider := "CxpCookieSessionManager"
    oDO:Timeout  := 15
  ELSEIF(cToken=="storage")
   oDO:Provider := "AppStorageManager"
   oDO:Store    := "cxp-data\application-storage.dbf"
  ENDIF
RETURN(oDO)
```

The file **CxpHttpServer.Prg** contains the source code for a web server which not only handles standard files such as .Htm, .Html, .Jpg, .Css, .Gif, etc., but it will also handle .CXP files. Most web servers are basically file servers which deliver content files to the web browser as they are requested. Sometimes, however, a web server is asked to do more than send back a file. Some extensions require processing on the server via an interface call CGI (common gateway interface). .CXP files require special processing and therefore they must be compiled into a .DLL and the .DLL is loaded by the <CXP/> runtime system.

The **CxpHttpServer.Exe** is started and is always listening on the designated port for connections from a client program, such as a web browser. This .exe is just like any other Xbase++ executable program with the exception that it includes the CXP runtime system.

There are many advantages to using your own custom server program in lieu of IIS or Apache.
Here are a few:

1. No other web server needs to be installed on your server.

2. The CxpHttpServer can be linked to your own libraries for access to your functions or your existing application. This simplifies setting up a CXP application that is very large. No helpers are required.

3. The CxpHttpServer can also handle debugging either locally or remotely. If you are an eXpress++ user you can use **WTF** or **WTFT** commands for debugging. WTF output will be displayed on the server. WTFT output will be sent to a trace client which can be anywhere on the internet provided that it can connect to the server. **Trace.Prg** is compiled into the CxpHttpServer.Exe program and **TraceClient.Prg** is compiled into a separate program named **TraceClient.Exe** which is started on a remote computer and connects to the server. When you use WTFT commands in your .CXP program, output will be sent to the remote trace program. For example, if the CxpHttpServer is running on donnay-software.com port 81, then the TraceClient is started like so: **TraceClient /PORT:81 /HOST:donnay-software.com.**

4. The CxpHttpServer can be as secure as you require. You can force it to only send out files of a specified extension.

5. The CxpHttpServer can perform other duties such as SOAP calls or delivering content based on simple RESTful URLs.

Conclusion

If you have a need to publish your data or your reports on the internet, whether for anonymous or specialized use, <CXP/> will greatly simplify the development and help you arrive at a satisfactory solution. It is worth the time to try it and see for yourself.