Managing Backup Preferences and Files

By Charles "Charlie" Vass, Technical Services Team Member, 4D Inc.

Technical Note 12-04

Table of Contents

Abstract

When developing a database application for distribution, such as a merged application without a data file, an added benefit would be to feature custom backup preferences in the application. The custom backup preferences could accompany the application at distribution, be configured with initial launch, or accommodate the preservation of existing backup preferences. This Technical Note shows how to create or configure a custom Backup.XML preference file or preserve an existing Backup.XML file for use with distributed applications independent of the native 4D Backup Preferences dialog.

Introduction

4D creates and maintains the application's backup preference file relative to the location of the structure file. This fact creates a situational challenge when it comes time to deploy a new application. This is especially true when deploying an update to an application that has an existing backup preference file and the data file is stored independently of the structure file.

The purpose of this Technical Note is to show how to manage backup preferences for distributed applications. It shows how to create a custom Backup.XML file or preserve an existing Backup.XML file.

This Technical Note includes a component that accomplishes the most sophisticated tasks in the creations and management of backup preferences. It also shows how to code a host application in a compiler friendly way to support optional inclusion of the component with a distributed application.

The Backup Preferences Management Problem

Building an application

When developing a 4D database that will be distributed to a customer, or within a diverse organization, the normal focus is on getting the main features of the database working. The testing of basic features, such as Backup for the merged application, is usually one of the last steps. This Technical Note will discuss issues and solutions for the often overlooked step of setting the backup preferences during the deployment of a distributed application.

Here is a typical scenario:

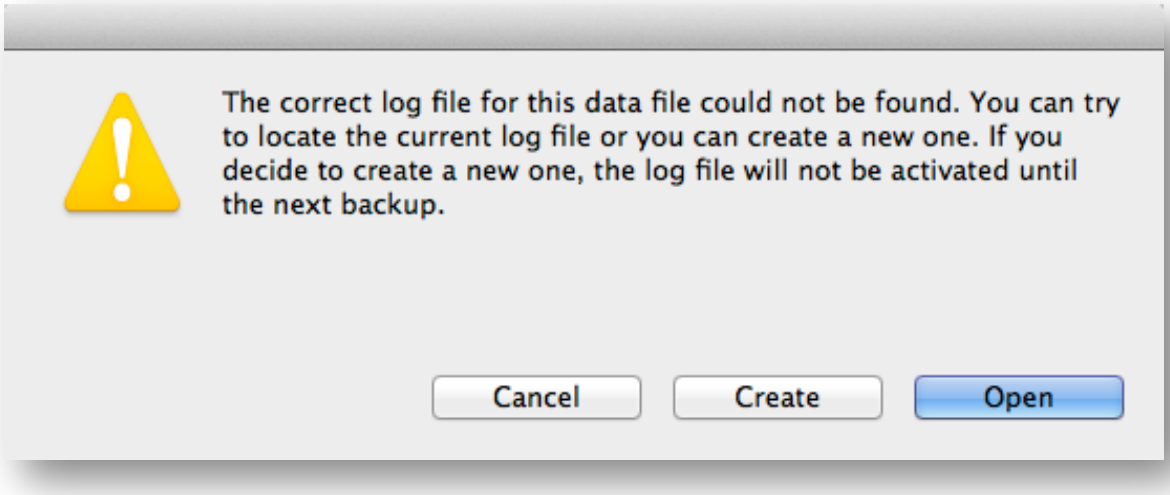
The database development is complete. The developer tests the application, including the Backup system. Everything seems fine. All the backup information is relative to the machine the testing was conducted on.

Note: By testing the Backup system a Backup.XML project file is created.

Now, the developer is ready to test the database as a built/merged Server application as it will be delivered to the customer. However, once the application has been built, not all files would have been copied inside the "Server" folder. Some of these files are under the "Preferences" folder. Therefore, the developer needs to copy some files from the "Preferences" folder to the new server's folder. However, the developer should not just copy the entire "Preferences" folder since it often contains unnecessary files like the XML file used to build the application.

At this point, the application seems to be working on the development machine. The next step is to copy the application onto another machine and launch it. Here is where the first problem is often encountered.

A folder cannot be found. 4D asks for the user to select a new folder path. What is going on here? How can this warning message be avoided?



The 4D Backup Project File

The problem comes when 4D needs valid backup preference settings. Preferences that tell 4D when, where, and what to backup. The Backup.XML file contains the when, where, and what for all files to be backed up. In the Backup.XML file:

- The *where* is now invalid because the newly deployed database has been moved to a new machine, and
- 4D asks for a new Backup path before the database can launch because the backup settings of the database or the 4D code in the database is configured to run a backup immediately.

So, how can this dialog be avoided?

One solution is to not place the Backup.XML file into merged applications, such as a built-server or standalone application. When the deployed application is launched and a backup is requested and there is no Backup.XML file, 4D will create a new one with all the default values. At this point, the backup may be launched.

Note: When a brand-new database performs a backup for the first time, 4D performs the backup and creates the default Backup.XML file in the "Preferences/Backup" folder, next to the structure file.

The default values may not match the current needs of the database because the developer might need to define some of the backup settings, such as:

- The number of archives to keep
- Extra files to be archived
- Advanced settings to speed up the backup process
- The specific backup schedule in mind

In this situation, where the developer provides his own predefined backup project file, users will still see the invalid backup path dialog. This is because the destination path specified in the Backup.XML may no longer be valid after moving the database.

Three different approaches to Backup.XML management

Copy and Restore the Backup.XML File

If the situation is that the existing Backup.XML file needs preservation, the most direct option is to copy the existing Backup.XML file whenever the database is closed and restore the file when the database is relaunched.

In the component database there is the method ***BackupXML_Archive,*** which is meant to be used in the On Server Shutdown and On Exit database methods. The first thing this method does is discover where the <database>.4DD file is stored. The assumption is that the data file will not be located with the merged application structure file. The method ***Get_4DD_Folder*** accomplishes this task:

```
  //  There's no constant to "Get 4D folder"
  //  that returns the path to the data file
  //  (
$4DDPath_T:=Data file
$Ndx:=Length($4DDPath_T)
  //  )

  //=======================  Method Actions  ============================

  //  From the end of the string, find the ending folder separator
```

```
  //  (
While (($4DDPath_T≤$Ndx≥#Folder separator)&($Ndx>0))
  $Ndx:=$Ndx-1
End while
  //  )

  //========================  Clean up and Exit  ============================

  //  Return the path to the <database>.4DD file
  //  (
If($Ndx>0)
  $0:=Substring($4DDPath_T;1;$Ndx)
End if
  //  )
```

After the path to the data file has been acquired, the Backup.XML file is copied to and save as a BLOB next to the data file:

```
If ($4DDPath_T#"")
  $BkUpPath_T:=Get 4D folder(Database Folder)+\
    "Preferences"+Folder separator+\
    "Backup"+Folder separator
  …
  DOCUMENT TO BLOB($BkUpPath_T;$BLOB)

  $4DDPath_T:= $4DDPath_T +"Backup.blob"
  If (Test path name($4DDPath_T)#Is a document)
    $Ref_H:=Create document($4DDPath_T)
    CLOSE DOCUMENT($Ref_H)
  End if

  BLOB TO DOCUMENT($4DDPath_T;$BLOB)
  …
End if
```

In the example database the method ***BackupXML_RestoreFile*** is meant to be used in the On Server Startup and On Startup database methods. Again, the first thing this method does is discover where the <database>.4DD file is stored. After that it transforms the BLOB back into a UTF-8 encoded XML file and saves it in the Preferences/Backup/ folder:

```
  //  Convert the blob back to XML
  //  (
DOCUMENT TO BLOB($4DDPath_T;$BLOB)
$XML_Ref_T:=DOM Parse XML variable($BLOB)
  //  )

If (OK=1)
   //  Restore the Backup.XML file in UTF-8 format
   //  (
  $BkUpPath_T:=$BkUpPath_T +"Backup.XML"
  DOM EXPORT TO FILE($XML_Ref_T; $BkUpPath_T)
   //  )
End if
```

Building a Backup.XML File

Given that the developer may need to specify their own values in the Backup.XML file, a better solution might be to create a custom Backup.XML file when starting the application. Using 4D's built-in commands to parse and write XML documents, the developer just needs to know the XML tags that will be used. The structure of these tags and generating the document is quite easy.

If the developer does not want the user to be involved in this process, they can create this document from the On Startup or On Server Startup database method of the application. When starting the database, the developer can check to see whether the Backup preference file exists or not by using the method ***BackupXML_Exists***. If the file does not exist, the developer may create a new XML file. The current user will not notice this operation. The backup will be ready to perform.

If the developer wants the user to be informed about the creation of the Backup.XML file and give them the ability to modify some settings, then a dialog could be displayed so that the user can change the settings. Based on the developer's hard-coded values and the user's last modifications, the developer can regenerate the Backup.XML file.

Note: The developer must be sure that the XML generated is well formed, e.g. that all parameters are valid or that there is not any extra space characters at the end of the values. Otherwise 4D Backup will generate its own default values for each malformed object.

Another Concept

Another concept is to have a Backup.XML file outside the application that will be used as a template. If the XML format changes, all the developer has to do is change this template XML file; there is no need to modify any 4D code, unless a new variable must be declared and initialized.

How does this work? The idea is to have a Backup.XML template file in which all the values are defined by 4D HTML tags. These variables are pre-defined in the 4D code. To keep things as basic and simple as possible, the 4D code does not use any XML commands; it just loads the template and executes the PROCESS HTML TAGS command, update for v12, and creates a new Backup.XML file in the "Preferences/Backup" folder for the database.

The XML template is provided with the component. Its name is Backup.XML2 and resides in the "Resources" folder.

Here is a summary of the steps involved in this technique:

- After building the merged 4D Server application, the developer is suppose to copy the "Preferences/Backup" folder to the "Server" folder. Instead of copying the Backup.XML file, install the BKP_Manager component.

- When launching the database, the On Server Startup method checks if a Backup.XML file exists.
- To get the path of the "Preferences" folder, use the Get 4D folder command and append "Preferences" + "Folder separator" + "Backup" to the folder path. Another technique would be to use the Structure file path instead. However, the Get 4D folder command is simpler and faster.
- Once the path has been computed, a call to Test path name on that path can be used to tell whether the Backup.XML file exists or not.
- If a Backup.XML file exists that means that the backup settings have already been set. The database is prepared and can continue.
- If the file does not exist there should be a new installation. Here, the developer has a few different choices on how to create the Backup.XML file:
    - The developer can make this completely invisible to the user. They can generate the Backup.XML file with their own pre-defined variables.
    - If the developer wants to give the user the ability to choose the folder right now or later, a request dialog can be used.
    - Finally, the developer might want to offer the user some direct control over the Backup settings. An "Advanced" backup settings dialog is provided in the component.

From the component, a simple dialog is displayed where the user can choose the backup destination folder. This dialog also contains a button that displays an Advanced Settings dialog that converts almost all options that the user can see in the default Backup Preferences dialog. Once done, the PROCESS HTML TAGS command is used to process the Backup.XML2 template and save the result as the Backup.XML file in the "Preferences" folder, more on this later.

The Backup.XML2 File

This file is just a default 4D Backup project with some modifications. All values have been replaced by 4D HTML tags such as 4DTEXT and 4DIF for when the values can be different.

Here is the list of all variables used in the template file (and also defined in the Advanced Settings dialog) with the corresponding XML key tag. For detailed documentation for each key refer to the 4D XML Key Backup PDF.

Variable Name	Tag Name	Values
DBNameItems_L	<DatabaseName><ItemsCount>	Integer

Variable Name	Tag Name	Values
DBNameItems_aT	<ItemsX>	Pathnames
LBPathItems_L	<LastBackupPath><ItemsCount>	Integer
LBPathItems_aT	<ItemsX>	Pathnames
LBLogItems_L	<LastBackupLogPath><ItemsCount>	Integer
LBLogItems_aT	<ItemsX>	Pathnames
CBSetItems_L	<CurrentBackupSet><ItemsCount>	Integer
CBSetItems_aT	<ItemsX>	Integer
LBDateItems_L	<LastBackupDate><ItemsCount>	Integer
LBTimeItems_aT	<ItemsX>	ISO DateTime
LBTimeItems_L	<LastBackupTime><ItemsCount>	Integer
LBTimeItems_aT	<ItemsX>	ISO DateTime
BKP_CB_RestoreLastBKP_L	<AutomaticRestore>	True or **False**
BKP_CB_IntegrateLastLog_L	<AutomaticLogIntegration>	**True** or False
BKP_CB_StartDbAfterRestore_L	<AutomaticRestart>	**True** or False
BKP_CB_IfModified_L	<BackupIfDataChange>	True or **False**
BKP_CompressionRate_aT	<CompressionRate>	**None**, Fast, or Compact
BKP_RedundancyRate_aT	<Redundancy>	**None**, Low, Med. Hi.
BKP_InterlacingRate_aT	<Interlacing>	**None**, Low, Med. Hi.
BKP_DelOldBKP_aT	<EraseOldBackupBefore>	**True** or False
	<CheckArchiveFileDuringBackup>	**True** or False
	<BackupJournalVerboseMode>	**True** or False
BKP_CB_KeepLastBKP_L	<Enable>	**True** or False
BKP_BackupSet	<Value>	Integer (Default is 3)
BKP_RB_AlwaysWaitBKP_L	<WaitForEndOfTransaction>	True or **False**
BKP_NbMinWait_L	<Timeout>	Integer (Default is 1)
BKP_RB_RetryNextTime_L	<TryBackupAtTheNextScheduledDate>	True or **False**
BKP_TimeRetry_aT	<TryToBackupAfter>	Hours, **Minutes**, Seconds

Variable Name	Tag Name	Values
BKP_CB_CancelRetryBKP_L	<AbortIfBackupFail>	True or **False**
BKP_NbTries	<RetryCountBeforeAbort>	Integer (Default is 5)
BKP_SegmentSize_aT	<DefaultSize>	**0**, 100, 200, 650, or 700
BKP_CB_StructureFile_L	<IncludeStructureFile>	**True** or False
BKP_CB_DataFile_L	<IncludeDataFile>	**True** or False
BKP_CB_AltFile_L	<IncludeAltStructFile>	True or **False**
BKP_BackupFileDest_T	<DestinationFolder>	Pathname
BKP_SA_Attatchments_L	<IncludesFiles><ItemsCount>	Integer
BKP_SA_Attatchments_aT	<ItemX>	Pathnames
BKP_RB_Sched_NoBKP_L		True or **False**
	<Frequency>	Hourly, Daily, Weekly, Monthly
BKP_RB_Sched_Hours_L		True or **False**
BKP_SchedEveryHours_L	<Hourly> <Every>	No. of hrs. (Def is 12)
BKP_SchedStartHours_aT	<StartingAt>	ISO DateTime
BKP_RB_Sched_Days_L		**True** or False
BKP_SchedEveryDay_L	<Daily><Every>	No. or Days (Def is 1)
	<Hour>	ISO DateTime
BKP_RB_Sched_Weeks_L		True or **False**
BKP_SchedEveryWeek_L	<Weekly><Every>	No. of Wks. (Def is 1)
BKP_CB_SchedEveryMonday_L	<Monday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_CB_SchedEveryTuesday_L	<Tuesday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_CB_SchedEveryWednesday_L	<Wednesday> <Save>	True or **False**

Variable Name	Tag Name	Values
	<Hour>	ISO DateTime
BKP_CB_SchedEveryThursday_L	<Thursday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_CB_SchedEveryFriday_L	<Friday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_CB_SchedEverySaturday_L	<Saturday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_CB_SchedEverySunday_L	<Sunday> <Save>	True or **False**
	<Hour>	ISO DateTime
BKP_RB_Sched_Months_L		True or **False**
BKP_SchedEveryMonth_L	<Monthly><Every>	True or **False**
BKP_SchedStartMonth_aT	<Hour>	ISO DateTime
BKP_SchedEveryMonthDay_L	<Day>	1, 2, 3… 29

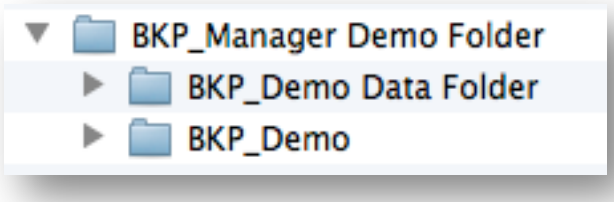
Component and Demo

This Technical Note includes a component and demo database. The BKP_Manager component was created to be used in one of two ways:

- As a one-time use component. Simply use the component once the database is ready for deployment and then remove it, or
- As an integral part of the database, making it easier for administrators to audit or make changes to backup settings.

Demonstrated in the startup and shutdown methods of the demo database is how to support compiler-friendly calls to the component if the decision is made to compile a database application for deployment and make the use of the component optional.

The environmental assumption for the demo database is that it is a database application that will be distributed to a client, and the data file is stored externally



from the structure package. The image below depicts the conceptual organization of the deployed database.

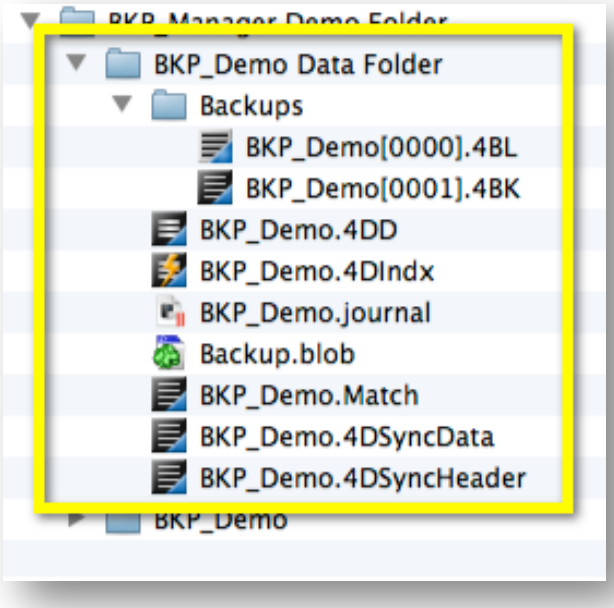
The Demo Data Folder

The primary purpose of demo database is to demonstrate options developers and administrators have when it comes to managing backup preferences. Having options in the management of backup preferences is really important when the data file is stored separately from the database structure file. In addition, this demo highlights a few significant protocols within 4D that a designer or administrator needs to be aware of when configuring backup preferences.

By default, 4D will back up the structure and data files as well as the ".4DIndx" file. The ".4DIndex" file is backed up as an attachment file. If "Replication" is enabled, it also includes the ".4DSyncData" and ".4DSyncHeader" files as attachment files.

When 4D adds the ".4DIndx" it will enter it in the "Attachments" list using a relative path such as "/.{databaseName}.4DIndx." This is because 4D assumes that the data file is stored with the structure file. This is not the case when Replication is enabled and ".4DSyncData" and ".4DSyncHeader" files exist.

The relative path to the ".4DSyncData" and ".4DSyncHeader" files, along with



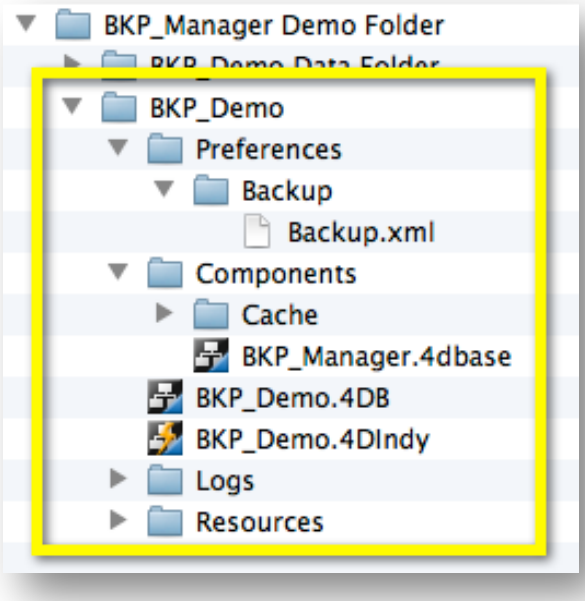
the ".4DMatch" and ".journal" files is based on the location of the data file and not the structure file. All these files are created and maintained by 4D in the folder that the data file exists in.

The above image is a screenshot of how the folder is organized; this contains the data file, BKP_Demo.4DD. In addition to the files 4D creates and maintains in the same folder as the data file, there is a folder titled "Backups" that is designated in the preferences as the destination folder for backup and a file named "Backup.blob" that will be explained in detail in the discussion of the component.

NOTE: Though all the files in the demo carry the same name as the structure file, if the data file has a name that differs from the structure file, all the files in the data folder shown above would have the name of the data file instead of the name of the structure file. The ramifications of this will be discussed in detail in the component discussion to follow.

The Demo Structure folder

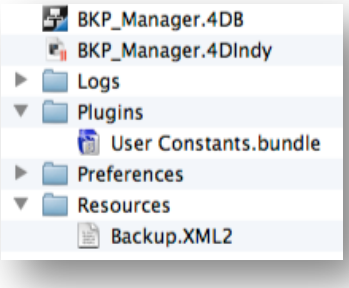
The demo structure folder, BKP_Demo, shown in the image below, conceptually represents what the folder of a deployed application structure would look like. It would have a few additional contents if it were a merged standalone application



or a built Client-Server application.

The BKP_Manager Component

The image on the right shows the contents of the BKP_Manager component package. The only



nonstandard item in the package is the Backup.XML2 file in the "Resources" folder. The purpose and contents of the file is described later in this section.

The User Constants plugin was created using the 4D Pop component and contains the constant Backup Prefs Folder, which is used in methods that need the On Server path to the "Preferences/Backup" folder.

Component Client-Server Interaction

The primary purpose of the BKP_Manager component is to make management of the Backup.XML file easy and portable. The component also provides an excellent study in writing a component that executes on a remote machine, while actively collecting and saving files stored on the server machine.

There are numerous methods in the component that have the "Execute on Server" property set. It is necessary for this component, while executing on a Client to copy file contents between platforms, test pathnames on the server, and to create and save files on the Server. To make it easy to reference which method has what properties, the properties for a method are listed in the header section of each method. An example header is shown below:

```
If (False)
  Begin SQL
  /*
    OnServer_Get_4D_Folder

    Purpose: Get the pathname of a folder on the server machine

    $0 - TEXT - Get the absolute local path on the server drive
    $1 - LONGINT - Type of folder
       $FolderType_L of Backup Prefs Folder (99) is Preferences/Backup folder
       If it does not exist, it is created

    Method Properties: Invisible / Shared with Host / Execute on Server
  */
  End SQL
End if
```

The coding style of using Begin/End SQL with SQL block comments "/* */" is documented in Tech Note Coding differently in 4D v12. The nice thing about this style is that it eliminates multiple lines beginning with double slashes (//).

Component Operational Concept

The operational concept of the component is to do three things;

- Archive the current Backup.XML file whenever the host application shuts down. This is accomplished by calling the component method BackupXML_Archive from Database Methods On Server Shutdown or from On Exit for a standalone host.

- Restore the Backup.XML file from the archived file whenever the host application starts up. This is accomplished by calling the component method BackupXML_Restore from Database Methods On Server Startup or from On Startup for a standalone host.

- Provide a custom interface and methodology for the creation of the backup preferences file, Backup.XML. This is accomplished by calling the component method M_CheckBackup from a host project method.

BackupXML_Archive

The ***BackupXML_Archive*** method is called from the host application's database shutdown or exit method to archive the current Backup.XML file. The Backup.XML file is archived into the same folder with the data file in a file named BackupXML.blob. If the data file is collocated with the structure file then another external folder (such as the Active 4D folder) should be chosen for BackupXML.blob file. The issue to avoid when selecting an external folder is that of selecting a folder that the host application does not have "write permission" for.

BackupXML_Restore

The ***BackupXML_Restore*** method is called from the host application's database startup method to restore the archived Backup.XML file to be the current Backup.XML file. It is especially important to do this at startup when deploying a new structure package to make sure that the local backup preferences are present before any scheduled backup.

M_CheckBackup

The ***M_CheckBackup*** method can be called from a host project method, object method, or menu method. It launches a new process for creating custom backup preferences.

The code snippet below shows the steps involved in creating the custom backup preferences. Following the confirmation to check the backup settings the steps are:

- Capture what type of host application is running, server or standalone

- Validate the pathnames in the current Backup.XML file

- Edit and save the custom backup preferences

```
CONFIRM("Do you want to check your backup settings?";"Yes";"No")
If (OK=1)
  $HostType_L:=Get_Host_AppType

    //  Validate the pathname entries in the Backup.xml file
    //  (
```

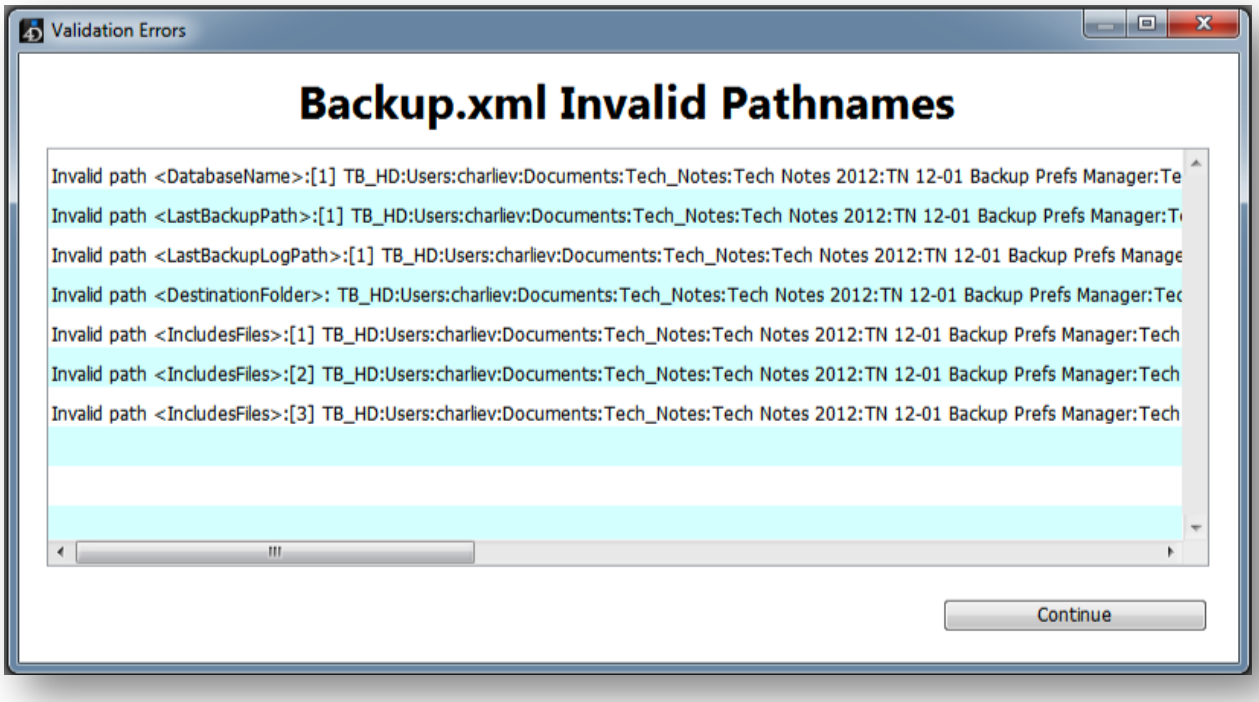
```
  ARRAY TEXT(ValidateErrors_aT;0)
  $Result_L:=BackupXML_Validate (->ValidateErrors_aT)
  If ($Result_L=0)
      //  Report errors in the Backup.xml file
      //  (
    $Ndx:=Open form window("ValidationErrors_d";\
    Movable form dialog box;\
    Horizontally Centered;\
    Vertically Centered)
    DIALOG("ValidationErrors_d")
    CLOSE WINDOW($Ndx)
      //  )
  End if
    //  )

    //=======================    Method Actions    =======================

  Case of
    : ($HostType_L=4D Server)
    M_BackupInit_Server

    : (($HostType_L=4D Local Mode) | ($HostType_L=4D Volume Desktop))
    M_BackupInit_Local

    Else
      ASSERT(False;"Unsupported 4D Mode")

  End case
End if
```

The method ***BackupXML_Validate*** loads the contents of the host's Backup.XML file and tests the validity of each pathname. Before the pathnames are tested for validity, each pathname is tested to see whether it is an absolute pathname or a relative pathname.

Pathname type check, absolute or relative, is performed by the component method ***BackupXML_Pathname_PreCheck***. The check is preformed because the command Test path name does not handle relative pathnames, it requires an absolute pathname. If the pathname is a relative pathname, it starts with "./," then the pathname to the data file is captured with the component method ***OnServer_Get_4DD_Folder*** so the absolute path can be created.

The leading "./" is stripped from the relative pathname and is then checked for the presence of the string "{databaseName}." The string "{databaseName}" is internal coding that 4D will sometimes places on the ".4DIndx" file as the first included file in the backup preferences. If this string is present, the command Replace string is used to replace it with the full pathname of the data file, minus the suffix "4DD."

After all relative paths have been converted to absolute paths; they are tested with the command Test path name. If the test fails, the pathname is captured in an array. After all pathnames have been checked whether there are any pathnames present in the array, the dialog shown below is displayed.



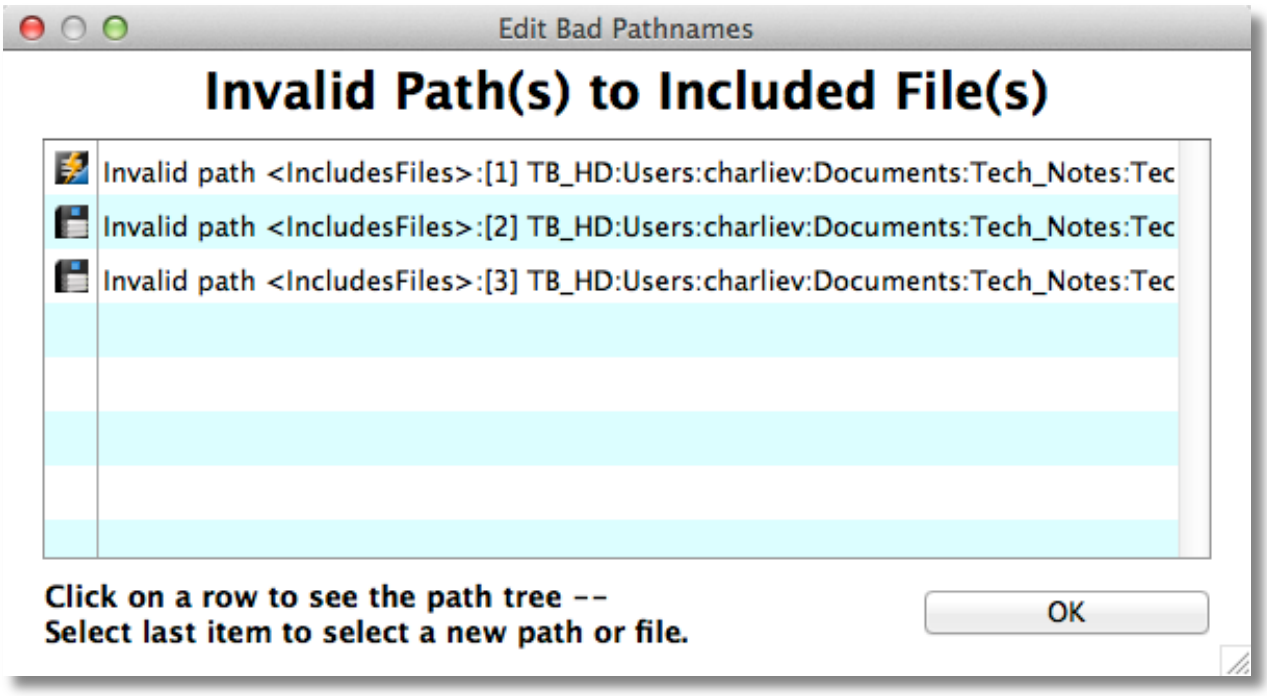
For all included files and folders, the component creates absolute pathnames for all attachments instead of relative paths that 4D will enter for selected attachments. Path strings can be long, long enough to extend beyond the viewing area of the list box. The complete path can be viewed by clicking on a row in the list box. A popup menu will appear, displaying the target file or folder at the top to the root of the path at the bottom.

Invalid paths can have one of the following headers:

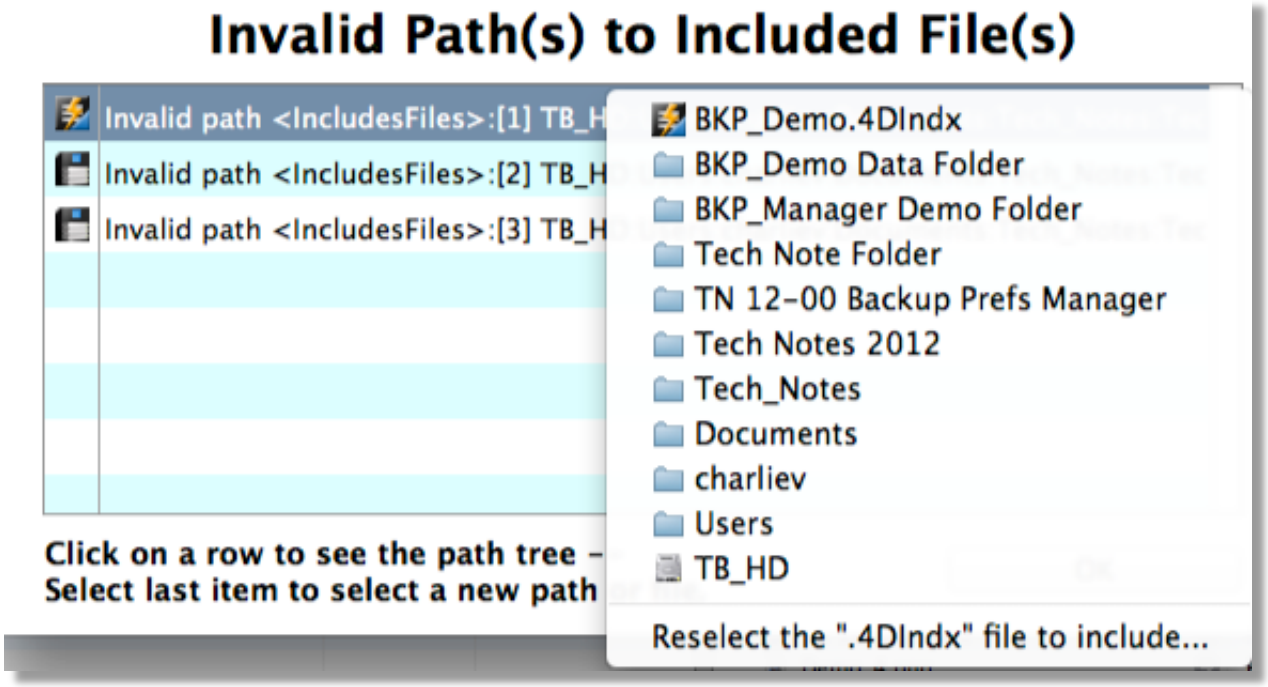
- Invalid path <DatabaseName>:[n]
- Invalid path <LastBackupPath>:[n]
- Invalid path <LastBackupLogPath>:[n]
- Invalid path <IncludesFiles>:[n]

Because these elements can have multiple entries, each line includes its item sequence number, [n]. The purpose of the sequence number is to make sure that the correct element is saved back to the Backup.XML file.

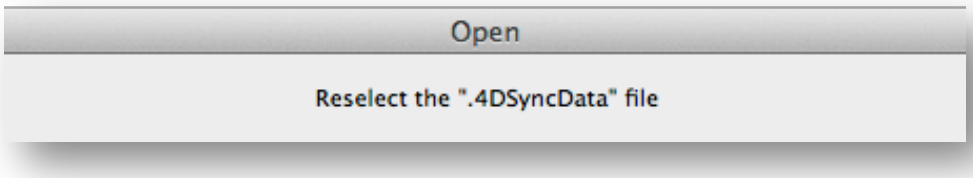
If the "Backup.XML Invalid Pathnames" dialog is presented with invalid pathnames, an editing dialog will be presented. The dialog, shown below, will be presented once for each type of invalid path that was present in the "Backup.XML Invalid Pathnames" dialog.



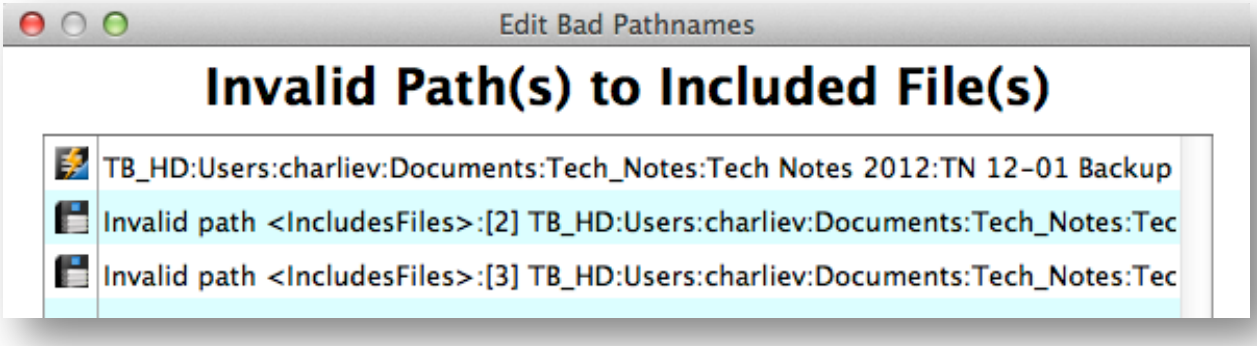
To correct an invalid pathname, a click on the invalid pathname will trigger the presentation of a popup menu that lists all the elements in the path, from the target file or folder to the root. See below:



The last item in the menu will trigger the execution of either the Select document or Select folder command. To help the user remember what document or folder needs to be selected, the appropriate prompt is present in the dialog, as shown in the next image.

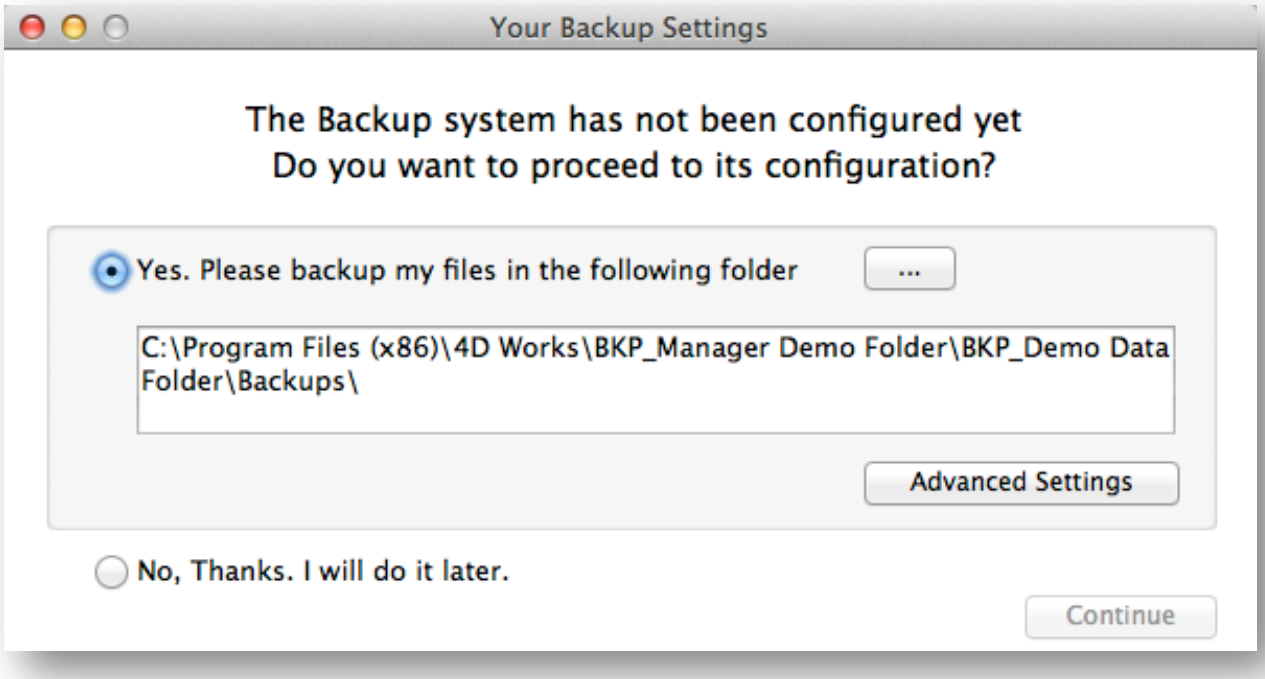


Once the new file or folder is selected, the invalid pathname is replaced by the new valid pathname, as shown below.



Once it is determined that all pathnames in the Backup.XML file are valid, the next dialog will appear (as shown below). This window allows users to edit the advanced backup settings and create the new Backup.XML file.

Note: If any pathnames have been corrected, they will only be applied to the Backup.XML file if the Advance Settings dialog is reviewed.

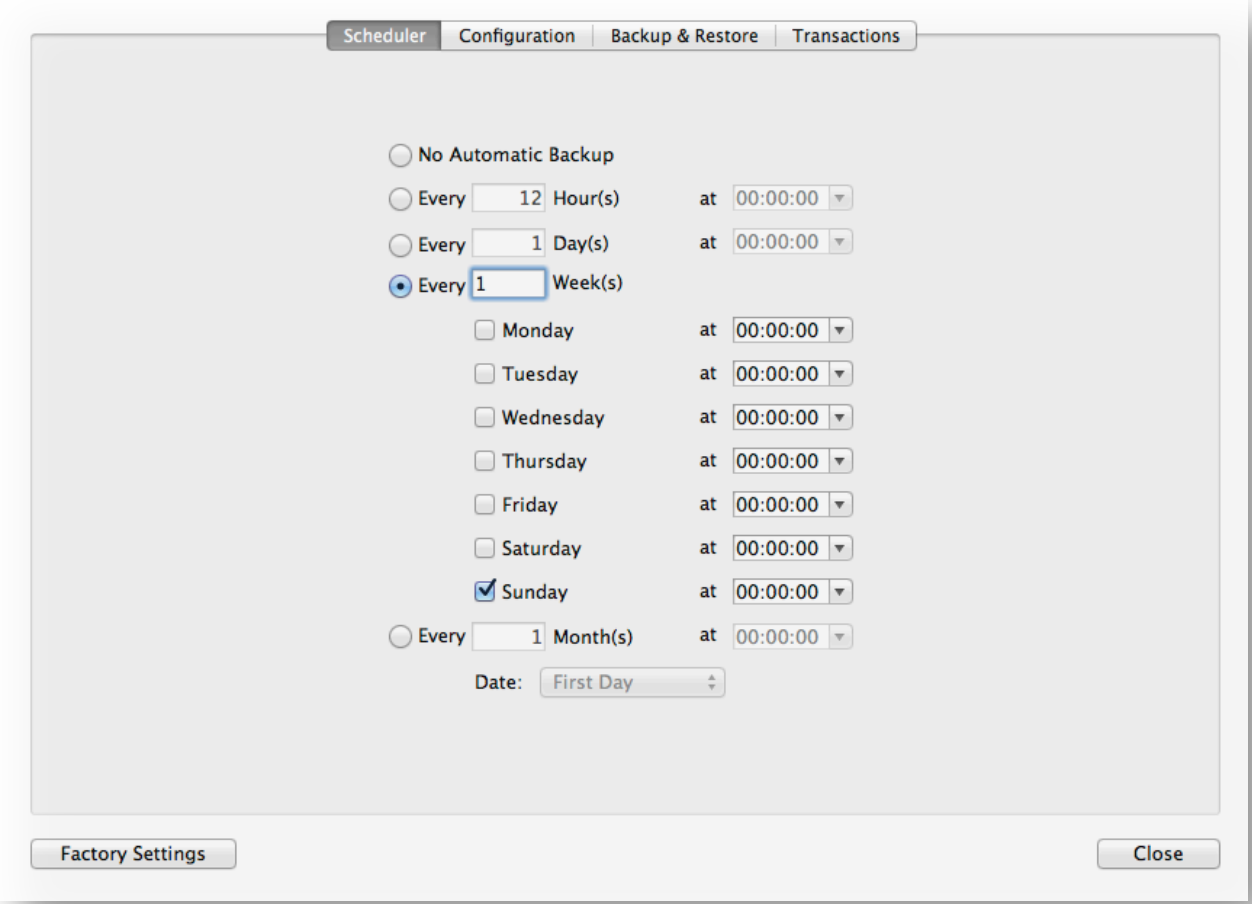


The "Continue" button will be enabled whenever the "No…" radio button is selected. When the "Yes…" radio button is active and the "Advanced Settings" dialog has been reviewed.

Advance Settings dialog

The "Advance Settings" dialog consists of four pages: Scheduler, Configuration, Backup & Restore, and Transactions. It contains all of the settings available from the "Backup Preferences" dialog, accessed by menu from Design/Database Settings…/Backup or by using the command OPEN 4D PREFERENCES.

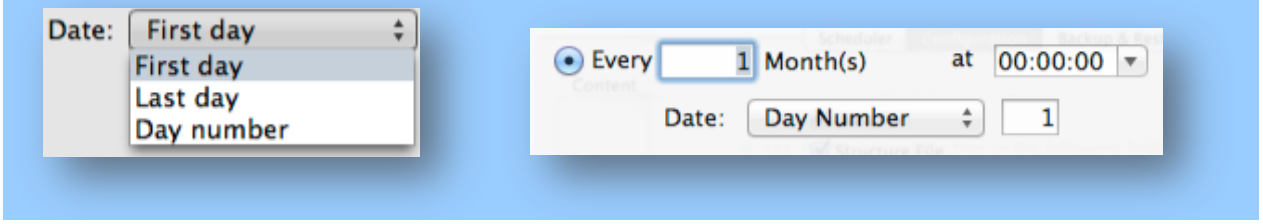
The **Scheduler** page, shown below, is used to set the *when* and *frequency* of performing backups.



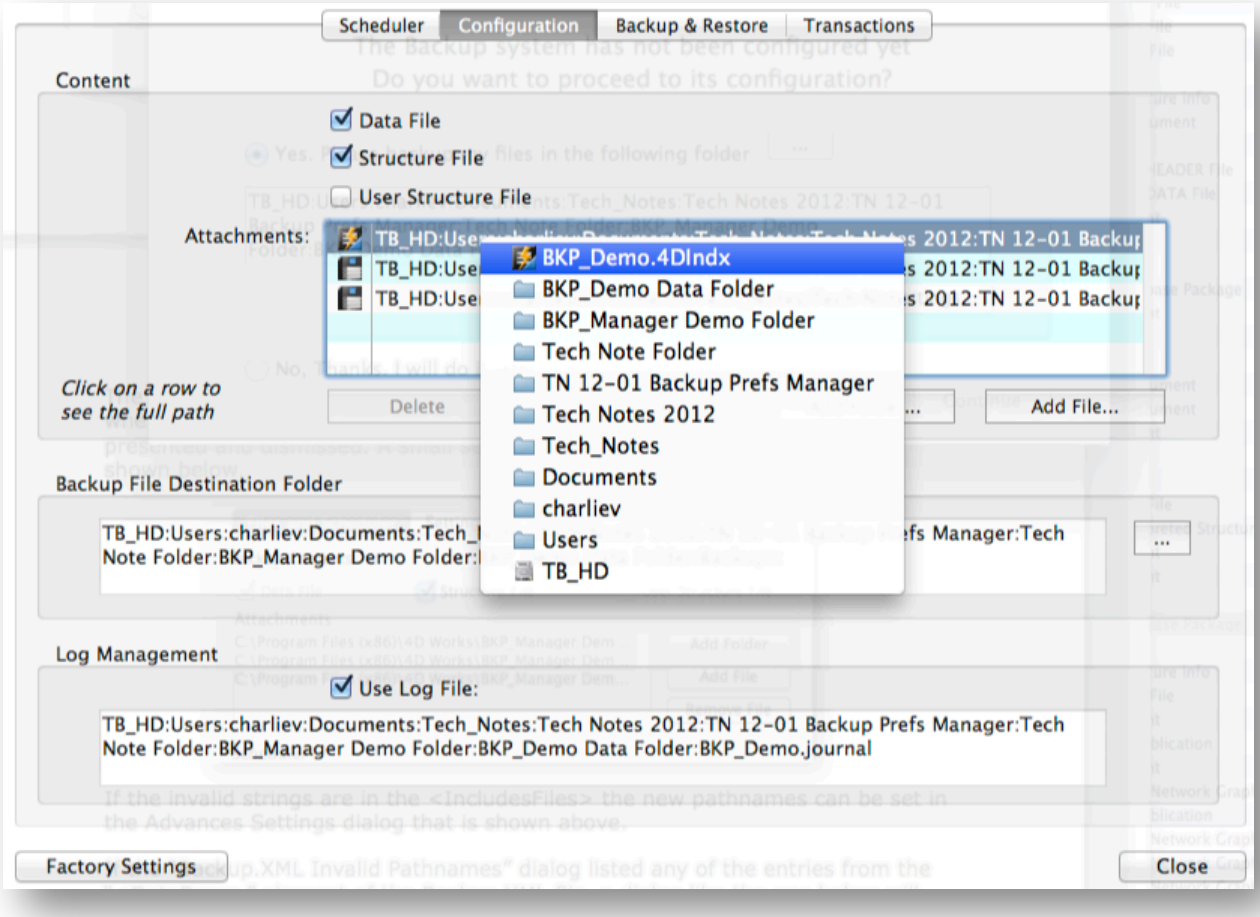
The only tricky part in setting *when* and *frequency* for backup is when the choice is "Every n Month(s)." The menu provides three choices: First day, Last day, and

Day number. When day number is selected an additional variable becomes visible to contain the day number of the backup.

Note: If Day Number is chosen, the maximum day that can be set is 28.



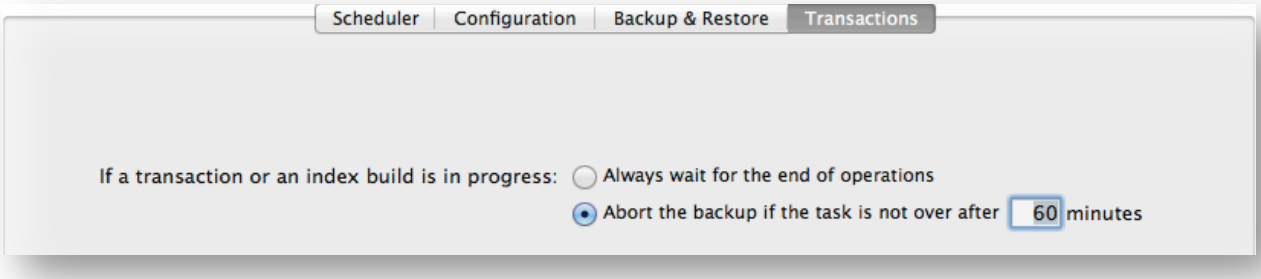
The **Configurations** page (shown below) provides for all the settings of the native dialog with the exception of the additional popup menu. The popup menu displays the complete path when the pathname is too long for full display in the



list box.

The **Backup & Restore** page is identical to that of the native preference dialog.

The component contains one additional page, **Transactions**. This page provides the preferences for what delay should be observed when transactions are active



or indexes are in the process of being built.

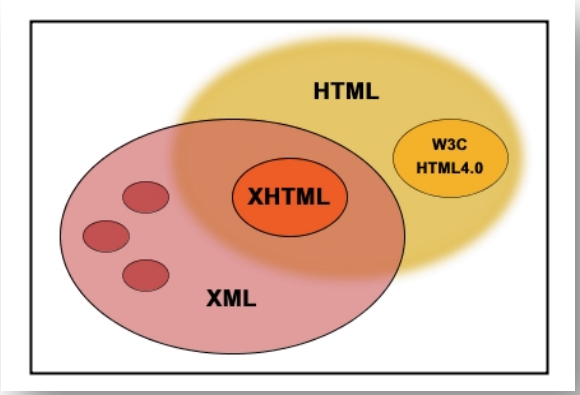
Processing the settings

Once the Advanced Settings dialog has been reviewed and dismissed, and the Continue button clicked, the Backup.XML2 file is loaded for processing.

```
$Path_T:=Get 4D folder(Current Resources folder)+"Backup.XML2"
...
  //  Load the marked up Backup.XML2 file
  //  (
$Ref_H:=Open document($Path_T;"utf8")
RECEIVE PACKET($Ref_H;$XML_T;Get document size($Ref_H))
CLOSE DOCUMENT($Ref_H)
  //  )

  //  Process the HTML tags
  //  (
PROCESS HTML TAGS($XML_T;$XML_T)
  //  )

  //  Save back to the server and archive it
  //  (
OK:=BackupXML_Transfer_LocToSvr ($XML_T)
If (OK=1)
   $Ndx:=Execute on server("BackupXML_Archive";512;"BackupXML_Archive";*)
End if
  //  )
```

The Backup.XML2 file is read from the Resources folder of the component into the text variable $XML_T. The next step is the unique and unusual action of this whole process. The command PROCESS HTML TAGS will replace all the variables imbedded within the XML file with the values from the variables of the same name that now reside in memory on the process stack.

So, how does a command intended for use with HTML files and is listed under the "Web Server" theme in the language come into play here? Simple, XML's purpose is to carry data; HTML's purpose is to display data. But, to display data is has to carry it. Though their purposes are different, the syntax of the two is the same. Beginning with HTML version 4, the XML rule of requiring the file to be "well-formed" was adopted. Well-formed simple means that the document conforms strictly to the published rules HTML syntax.



That is where HTML and XML overlap. The markup of the language is so similar that the command PROCESS HTML TAGS does not distinguish one from the other. The overlap in the two languages provides the opportunity to use the 4D command in a very innovative and productive way.

Once the tags have been processed, the file is then written out as follows:

```
$Ref_H:=Create document($Path_T+"Backup.XML";"utf8")
If (OK=1)

    SEND PACKET($Ref_H;$XML_T)
    CLOSE DOCUMENT($Ref_H)  // Close the document
    $0:=OK

End if
```

Notice that when the XML files are opened and created, the file type of "utf8" is used. Files saved as XML must be in the "utf8" format.

Summary

This Technical Note showed how to manage backup preferences when distributing a new merged application. It showed how to create a custom Backup.XML file and how to preserve an existing Backup.XML file which is to be used by the 4D Backup system with distributed applications.

As an added feature, this Technical Note includes a component that can be used to accomplish the most sophisticated tasks in the creations and management of backup preferences.

Conclusion

When creating a standalone or built client-server application for distribution sans data file in 4D v11 SQL, 4D v12, or 4D v13, it may be desirable to be able to accommodate custom backup preferences. These preferences may accompany the application or accommodate preservation of existing backup preferences of an installed system. This Technical Note showed multiple methods on how to create a custom Backup.XML file and how to preserve an existing Backup.XML file. This Technical Note also demonstrated how to write a component that executes on a remote client and collect and saves files on a server machine.