CodeExport Component

By Josh Fletcher, Technical Account Manager, 4D Inc.

Table of Contents

Introduction

Revision control is a fundamental part of software development and yet, because of 4D's propensity to use binary files, it remains an unsolved problem for many 4D developers.

The CodeExport component facilitates automatic export of all methods in the host database to text files on disk. If the host database is under revision control, these text files are suitable for committing to the repository, thus giving the developer to ability to track changes to methods over time.

Most importantly, the CodeExport component is designed to have **zero impact** on the host database. There is no startup code to install and nothing to configure.

Installation

Note: *CodeExport requires 4D v13.*

Install the component in the "Components" folder if your database. No further steps are necessary.

Optional Installation

Though not necessary, startup code can be added to launch the component at database startup (see next section). There is a macro included to add this code called "CodeExport: Startup Install - Optional". The output of the macro is the following code:

```
C_LONGINT($found_l)
ARRAY TEXT($components_at;0)
COMPONENT LIST($components_at)
$found_l:=Find in array($components_at;"CodeExport")
If ($found_l>0)
   EXECUTE METHOD("CE_OnStartup")
End if
```

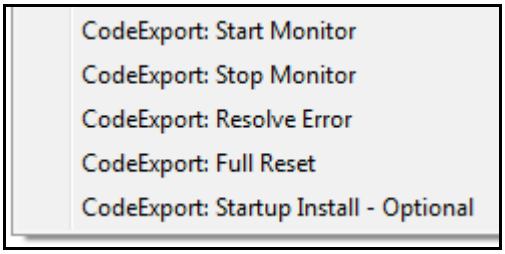
Note: *The CodeExport component is interpreted. The Source Toolkit feature cannot be used in compiled mode. There is no reason to use this component in a compiled database.*

Usage

With the component installed, launch the database and open a method (or if one was already open, the startup code runs as well).

Using macro events, a stored procedure named "CodeExport Component Process" is launched to export all the code in the database and continues to run in the

background. You can stop and start the process via macros (look in the macros menu):



Note: *If the component estimates that the export may take longer than 1 second, a progress bar is displayed.*

The component creates two folders next to the database:

- **ce_data** – this folder contains an external database that the component uses to track metadata about the host. This external database is stored outside of the component so that the component can be upgraded without losing the metadata.
- **ce_source** – this folder contains all of the exported methods.

The stored procedure checks for method changes every second, and exports them if needed.

Note that if the component is installed in a client-server database, the stored procedure runs on the server (thus the code is exported on the server).

Features

This section summarizes the main features of the CodeExport component.

Method Export

Using the Source Toolkit feature set of 4D v13, CodeExport creates UTF-8 text versions of every method in the database.

Note that while Project methods are exported to the root of the "ce_source" folder, all other method types are exported to subfolders as needed (table form methods, for example, are placed in a folder with the form name inside another folder with the table name).

The first time the component is enabled, every method in the database is exported. This can take some time in larger databases. Thereafter only those methods that are changed are exported (along with new methods of course).

Adaptive Progress Bar

In addition to exporting code, CodeExport validates method names (this is necessary to avoid exporting invalid methods). Both of these operations can be time consuming depending on the number of changes since the last export. CodeExport attempts to estimate the length of any such operation and, if it is estimated to take longer than 1 second, a progress bar is displayed.

Macro Driven

CodeExport is driven completely by macros. The benefit of this design is there is no impact on the host database. There is no component code that needs to be called and nothing to configure.

To get the code export started, Method Editor macro events are used. If the user opens, closes, saves, or creates a method, a hidden macro is called. This macro executes the component startup code automatically.

Bear in mind there is normally no need to explicitly start the component up. CodeExport only exports methods. Opening a method to work on it automatically starts the component (as does leaving a method open such that it opens when the database is restarted).

The following macros are available to the user:

- **CodeExport: Start Monitor** – this macro starts the stored procedure that exports the code. Note that it is **not** necessary to call this macro under normal circumstances. This macro is useful if the process was explicitly stopped.
- **CodeExport: Stop Monitor** – this macro stops the stored procedure. Again there should be no need to call this macro under normal circumstances; the process is capable of detecting a database quit, for example.
- **CodeExport: Resolve Error** – if the stored procedure encounters an error, the error is displayed but the process continues executing. It simply skips the code export. If the error is later resolved by the user, execute this macro in order to restore the code export.
- **CodeExport: Full Reset** – this macro forces a full code dump and also deletes an error log data.
- **CodeExport: Startup Install – Optional** – this macro can be used to install optional code to force the component to launch during database startup.

Regarding "CodeExport: Startup Install – Optional", it can be desirable to explicitly launch the startup code in some situations. The first presumption is that when the database is open, a Method Editor is never opened. This is obviously a rare occasion during development.

Secondly a code-affecting change needs to occur that does not involve opening a Method Editor window. A global find and replace (GFR) is a good example. If the developer starts the database, performs a GFR, and quits, those code changes would not be exported; at least not until the next time a Method Editor is opened.

Error Handling

CodeExport is somewhat pessimistic in its error handling. If an error occurs, the component disables itself. The stored procedure continues to run, but will not export any code.

The error is only shown to the user once (to avoid any issues with looping on the same error).

Likely reasons for this kind of error would be file system problems when exporting the methods or SQL errors when accessing the external database.

Once the problem is resolved, the macro "CodeExport: Resolve Error" should be used to restore the export function. Alternatively simply restart the database.

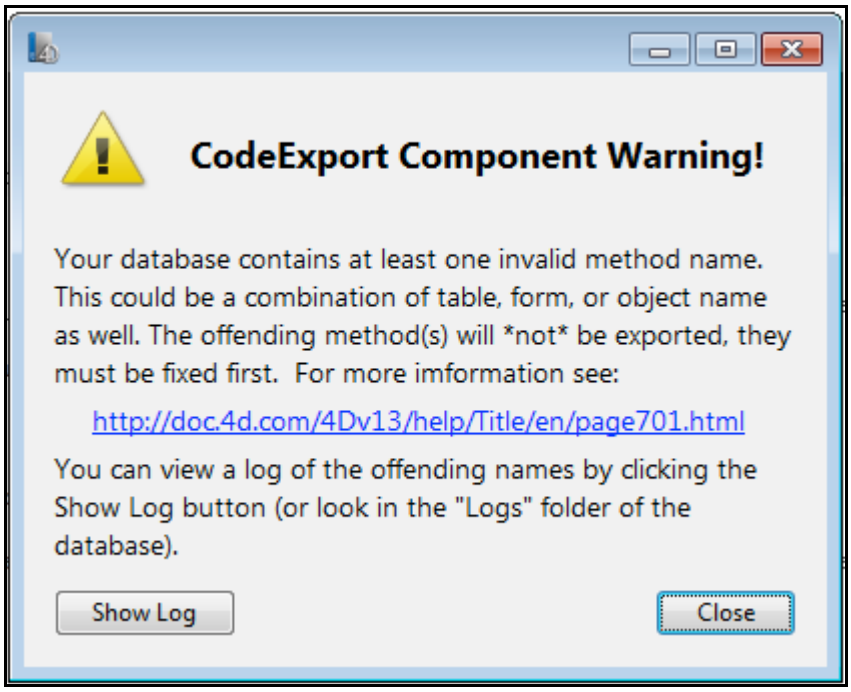
Method Name Validation

4D v13 is more stringent about method names than previous versions of 4D. In particular method names must:

- Begin with an alphabetic character or underscore.
- Thereafter, the name can include alphabetic characters, numeric characters, the space character, and the underscore character.
- Periods, slashes, quotation marks and colons are not allowed.
- Characters reserved for use as operators, such as * and +, are not allowed.
- 4D ignores any trailing spaces.
- See http://doc.4d.com/4Dv13/help/Title/en/page701.html

Similarly the paths returned by the Source Toolkit are intended to be compliant with the file system and follow these same rules. As such the possibility exists that converted databases contain methods with invalid names. These methods cannot be exported to disk until they are fixed.

If the database contains invalid method names a warning is displayed:



A log file containing the invalid method names is created in the "Logs" folder of the host database as well. This file is named "ce_errors.txt".

CodeExport will continue to ignore these methods until they are fixed.

Design Highlights

This section highlights some of the important pieces of the CodeExport component's design. These highlights may put the spotlight on a certain 4D features, or just point out design decisions from the component. These sections are sorted with the most important items listed first.

Database Organization/Naming Convention

CodeExport is broken into several different modules using the following naming convention:

- Modules are noted by capitalized tokens separated by underscores. For example the method "CE_LOG_ShowWarning" can be read as the "ShowWarning method of the CE_LOG_ module".
- All component-specific code is prefixed with "CE_" (meaning "CodeExport").
- Methods without the "CE_" prefix are generic enough to be used outside the component, feel free to copy them.
- After the module name, items in camel case are method names.
- All variables have the data type as a suffix, i.e.:

- _t is Text
- _l is Longint
- _f is Boolean (“f” for flag)
- _b is BLOB
- _p_@ means pointer to the type that follows
- Etc.

METHOD GET PATHS

The core of CodeExport is the Source Toolkit command “METHOD GET PATHS”. Without this command the component would not be possible. In particular this command offers 3 main features:

- Returns an array of paths to methods.
  - Every method in the database has a unique path (Source Toolkit guarantees this).
- The path returned complies with the file system (unless the method name is invalid).
- Accepts a stamp as input and returns the current database stamp as output.
  - This allows the caller to get the paths to only the changed methods, or all methods by passing 0. Note the value passed must be a numeric variable because it is used for input and output, it cannot be a literal.

METHOD GET PATHS can be used to get the path to methods in either the host database or the component. CodeExport only accesses methods in the host, of course, unless executed in Matrix mode.

METHOD GET PATHS returns paths that are in POSIX format. Additionally any special characters are encoded. In other words to use a Source Toolkit path to create a file on disk, it must be converted. The method “CE_Method_Save” accomplishes this as follows:

```
$methodPath_t:=$1
$filePathFragment_t:=Convert path POSIX to system($methodPath_t;*)
```

The command “Convert path POSIX to system” is used to convert the Source Toolkit path to something the system understands. Note that the same code works on Mac OS X or Windows. Finally the * parameter is used because the Source Toolkit paths are encoded.

Validating and Saving Methods

Several important 4D features are used to validate and save the methods to disk.

Method validation is achieved by evaluating each part of a method path for compliance with 4D’s rules for identifier names. More specifically:

- The command METHOD RESOLVE PATH breaks a Source Toolkit path up into its component parts as appropriate:
  - Method name
  - Form name (if form or object method)
  - Table name (if trigger or table form)
  - See "CE_Method_Validate"
- These parts are each evaluated using MATCH REGEX to verify conformity with 4D's identifier name rules. See the method "UTIL_IsIdentifier".

The METHOD GET CODE command accepts a Source Toolkit path and returns the source code of the method, if it exists. Additionally the code returned contains a special comment line at the top with all non-default Attribute values. Here is an example:

```
//%attributes = {"lang":"en","invisible":true} comment added and reserved by 4D.
```

With CodeExport and revision control, changes to attributes can be tracked in addition to the code itself.

With the method code in hand, and a path to that method, several other 4D features play an important role:

- The CREATE FOLDER command accepts the * parameter in v13, allowing it to create any missing folders for the target path.
  - With the new behavior of CREATE FOLDER, you can call this even if all the folders already exist, simplifying the code. There is no need to test for the folder.
- The code returned by METHOD GET CODE is "4D Text", which is always UTF-16. The exported files are UTF-8 because this is more common and the de facto standard for text files. To perform the conversion, the CONVERT FROM TEXT command is used. See "CE_Method_Save".
- To ensure the files are correctly interpreted as UTF-8, a Byte Order Mark (BOM) is inserted at the beginning of each file. See this Tech Tip for more details: http://kb.4d.com/assetid=76555

External Database

CodeExport uses an external database to track metadata about the host database. In particular the external database ("CE_Data") contains two tables:

- **CE_Prefs** – this table tracks the latest stamp value for the host database (but can be extended to support other preferences, it is just a name/value pair table).
- **CE_BadPathLog** – this table tracks information about methods in the database with invalid names. This data is used to generate the log file.

The power of the external database feature in this case is that it allows the component to have its own database without affecting the host application. In

addition the external database is simply a 4D database. To inspect the data contained in CE_Data, open it in 4D.

External database access is achieved using the "USE DATABASE" SQL command, for example:

```
$statement_t:="USE REMOTE DATABASE DATAFILE '"+<>CE_DataFile_t+"'"
Begin SQL
EXECUTE IMMEDIATE :$statement_t;
End SQL
```

In order to support development with 4D Server, CodeExport actually also uses the "REMOTE" keyword to ensure the external database is always opened on the server.

It takes some time and resources to open external databases. In fact 4D already optimizes this on a global level. When a process opens the external database, 4D creates the objects necessary to access it. It does not, however, recreate the same objects if another process opens the external database. Similarly when the connection is closed (via "USE DATABASE" with "DEFAULT") 4D does not automatically destroy the connection objects. This can be forced by opening the connection with the "AUTO_CLOSE" option, but this is not recommended for best performance.

Even with 4D's external database connection optimization, performance can be further increased by not calling USE DATABASE unless absolutely necessary. CodeExport takes optimization one step further:

- Per-process, USE DATABASE is only executed if a connection is not already open.
  - Otherwise a counter is increased.
- When a close request is made, the counter is decremented.
  - If the counter reaches 0, then the connection is actually closed.
- This offers two advantages:
  - Code accessing the external database runs faster.
  - Nested connections can be made. Said another way, certain CodeExport methods require an external database connection. These methods are free to open and close connections as needed without worrying whether or not a connection is already open.
- See "CE_EXT_Open" and "CE_EXT_Close" (and their callers) for more details.

Note that AUTO_CLOSE can be useful in some contexts. Under default circumstances (without AUTO_CLOSE) the external database cannot by opened by another 4D instance, even if all connections to it are closed. With AUTO_CLOSE the external database is available for outside use even while the main database is running. This is mainly useful for troubleshooting and/or

development, so that the contents of the external database can be observed in real-time (e.g. with another copy of 4D).

Macros

As mentioned previously, CodeExport is macro driven. The important fact to notice is that components can contain their own macros and those macros are accessible from the host database. See the "Macros v2" folder inside the CodeExport component to view the macro file.

One additional feature that CodeExport takes advantage of is the ability to hide macros. Macros have several XML attributes to facilitate this:

- **type_ahead** – this attribute allows the developer to hide or show a macro in 4D's type-ahead list. All macros in CodeExport have this set to "false".
- **in_menu** – this attribute allows the developer to hide or show the macro in the "Insert Macro" context menu item when right-clicking in the Method Editor. All macros in CodeExport have this set to "false".
- **in_toolbar** – this attribute allows the developer to hide or show the macro in the toolbar macro list in the Method Editor. Macros made available to the user in CodeExport have this set to "true", while hidden macros have it set to "false".

Quit Detection

The CodeExport stored procedure is "self-quitting". It automatically detects if the 4D database is quitting and gracefully closes itself (thus there is no need of On Exit code).

This is accomplished via a simple 4D rule: when the database is quitting, calls to DELAY PROCESS have no effect. Since the stored procedure normally delays itself 60 ticks per execution, if the delay is less than 60 ticks then the database is quitting. See "CE_MON_DelayProcess".

Progress Bar

CodeExport takes advantage of the new 4D Progress component included in 4D v13. This component offers platform standard progress bars on both Mac OS X and Windows. See http://doc.4d.com/4Dv13/help/Title/en/page3058.html for more details.

CodeExport goes one step further by enhancing 4D Progress to only show a progress bar if an operation is estimated to take longer than 1 second, otherwise the progress bar is not shown. This is done to improve performance. Most of the time (after the initial code dump) CodeExport will only be exporting small sets of methods so there is no need to have a progress bar.

The algorithm used is quite simple. An estimate is computed based on the progress made so far and the time it has taken so far. See the method "PROG_ProgressRequired" for more details.

Error Handling

CodeExport contains a generic error handling module ("UTIL_ERR_") built around a few rules:

- Each error is defined by an interprocess Longint value (see "UTIL_ERR_Startup").
- No two modules have overlapping error values, all error values are unique.
- Error values do not overlap with 4D errors.
- Each error value has a corresponding message (see "UTIL_ERR_GetErrorMessage").
- Any error dialog displays any and all error data; nothing is hidden nor needs to be expanded (see "UTIL_ERR_HandleError").
- All error dialog data can be copied to the clipboard.

This error handling module is very much meant to be a developer tool; it is not "user-friendly" and attempts to display as much "gory detail" as possible. The reason for this is to make it as efficient as possible to track down errors. CodeExport is a development tool and should NEVER be deployed to a customer.

This module preserves and restores any existing error handlers as well (see "UTIL_ERR_HandlerInstall" and "UTIL_ERR_HandlerRemove").

UTIL_ methods

There are three generic methods in CodeExport that may be of use in any 4D database.

- **UTIL_IsIdentifier** – use this method to test if a name conforms to 4D's rules for identifier names. This can be used for table names, form names, variable names, method names, etc.
- **UTIL_IsComponent** – use this method to detect if code is executing in a component or host/matrix database.
- **UTIL_MethodExists** – use this method to test to see if a method still exists in the database. Note: requires Source Toolkit path to the method.

Conclusion

More than anything else the goal of this component is to convince every single 4D developer to strongly consider trying revision control with 4D, while having no impact on their existing development practices. From solo developers to large teams, the benefits of revision control are immediately apparent. The CodeExport

component makes the first step even easier by automatically exporting all of the database methods. There is nothing to lose, and so much to gain.