Omnis Studio 8.1 BETA

JSON Library Representation

OMNIS STUDIO 8.1 BETA USE ONLY.

Document Overview And Scope

This document describes the JSON library representation in Omnis Studio 8.1.

History

Issue	Date	Author	Status	Reason	Distribution
1	10-Jan-17	Bob Mitchell	First Draft	Initial document	R&D

Table Of Contents

1 INTRODUCTION

Omnis Studio 8.1 allows you to export an Omnis library to a directory tree containing JSON and other files that contain the contents of the library in a human readable form. Additionally, 8.1 also allows you to import an Omnis library from such a directory tree.

Providing this feature is important as it allows developers to use 3rd party version control systems such as GIT and SVN to manage source code, and in particular this then provides a good way to encourage the sharing of Open Source Omnis libraries.

This document describes the starting point that has been added to 8.1: notation calls that export and import the Omnis JSON library representation.

2 TREE STRUCTURE

Note - all text files use UTF-8 encoding, and are formatted nicely for viewing in a text editor.

2.1 General

This section describes the directory and file structure of an exported library.

A library is represented by a folder that contains the file library.json. We call this folder the library folder. library.json contains top-level information about the library e.g. the library preferences.

Within the library folder, there is a tree of class directories that represents the folder structure of the Omnis library. Each class has its own directory, and if the class itself is an Omnis folder class, it contains sub-directories for the Omnis classes contained in that Omnis folder.

Each class directory is named with a name created using the class name (see the note on directory and file naming below).

Every class directory contains a JSON file named class.json. This contains top-level information about the class, including:

- Class type
- Class properties
- For classes that support methods: definitions of class and instance variables, and for task and remote task classes, definitions of task variables.

File classes also have a file called indexes.json within the class directory, if the file class defines indexes.

If the class supports methods, the class directory also contains a JSON file named methods.json provided that there are some class methods. methods.json contains an array of the class methods, where each entry contains various properties of the method and definitions for parameters and local variables.

There is a file in the class directory for each method defined in methods.json, named <method name>.txt (subject to the file naming rules below), that contains the method code.

If the class can contain objects, then there are 2 different structures depending on the class type:

- For file, query, schema and search classes, all objects and their properties etc. are in a single file called objs.json in the class directory. objs.json contains an array of objects.
- For all other class types that can have objects, the class directory can have a number of sub-directories:

- objs
- bobjs
- inheritedobjs

The obj directory contains a sub-directory for each object in the class, where the directory name is the object name (subject to the directory naming rules below). Each object sub-directory contains a file named object.json that contains object properties etc, and if the object has methods, there is an identical structure to that used for the class methods: a methods.json file, and method .txt files.

The bobjs directory is only present for window classes. It contains a sub-directory for each background object in the class, named using the object ident (subject to the directory naming rules below as older libraries can unfortunately contain objects with duplicate idents). Each background object sub-directory contains a file named object.json that contains object properties etc.

The inheritedobjs directory is only present for classes that support inheritance. It contains a sub-directory for each superclass object that either defines or overrides a method in the subclass. Each sub-directory contains methods.json and method .txt files just like those used for class and object methods, representing the methods defined or overridden for the object.

2.2 Binary Data

There are a number of properties which require a binary representation in the JSON library representation. We handle these in 2 ways:

1. If Omnis recognises a PNG e.g. in #ICONS or a report background picture, it outputs a PNG file to the tree, and the JSON contains the name of the PNG file.
2. Otherwise, Omnis outputs the BASE 64 encoding of the binary data to the JSON file.

2.3 Directory and File Naming

Where possible, directories and files are named using the Omnis name (class name, object name, object ident, or method name). However, there are some considerations:

1. Although it is not necessarily sensible, Omnis names can contain characters that are not allowed in file system names e.g. path separators for all platforms, ?, *. To cater for this, the JSON library representation escapes these characters as % followed by the 2 lower case hex characters that represent the escaped character. As a consequence, Omnis also escapes the % character.
2. Omnis libraries can contain classes where the names only differ by their case. In addition, they can contain objects with duplicate names. In these cases, the JSON library representation prefixes the name with the string %_<n>_ where <n> is an integer index (for objects this is the order value, and for classes this is a value starting at 1 and incremented for each class with the same case-insensitive name; note that Omnis always exports classes in ascending name order, meaning that the prefix for each class in a set of classes with the same case-insensitive name will be the same each time you export the classes, unless you add or remove a class with the same case-insensitive name).

3 NOTATION

3.1 Export JSON

$root.$exportjson(rLib, cOutPath, &cName [,&lErrorList, &lWarningList])

Exports JSON tree for library. Parameters:

- rLib is an item reference to the library to export.
- cOutPath is the pathname of the directory in which $exportjson will create a new directory containing the JSON library representation.
- If $exportjson succeeds then cName receives the name of the new JSON library representation directory created by $exportjson. The name is based on the name of the exported library and the current date and time. Note - cName is a name, not a pathname.
- lErrorList and lWarningList are lists that receive errors and warnings about the export process. $exportjson defines these lists, so there is no need to define or clear the parameters before calling $exportjson.

$exportjson returns kTrue for success, and kFalse for failure.

In the case of failure, lErrorList contains error reports, and $exportjson has cleaned up by removing any partially output JSON library representation before it returned.

In the case of success, lErrorList is empty. lWarningList may contain various warnings about the export process e.g. duplicate object idents or object names.

In addition, certain errors or warnings contain a note that there is an entry in the Find and Replace log, which allows the developer to go straight to the source of the problem.

The error list and warning list each contain 3 columns:

- class: Item reference to the class for which the error or warning is being reported.
- errorcode: Unique integer error code for the error or warning.
- errortext: Error text corresponding to the errorcode.

$exportjson displays a working message if it executes for more than a second, and this allows the user to cancel the export, in which case $exportjson returns kFalse and adds error 23433 to the error list.

3.2 Import JSON

$root.$importjson(cJsonFolder, cLibPath [,&lErrorList, &lWarningList])

Imports JSON library representation. Parameters:

- cJsonFolder. The pathname of the JSON library representation directory. $importjson validates this by checking for the presence of library.json in this directory.
- cLibPath. The pathname of the new library to be created from the input JSON library representation. This file must not already exist.
- lErrorList and lWarningList are lists that receive errors and warnings about the import process. $importjson defines these lists, so there is no need to define or clear the parameters before calling $importjson.

$importjson returns kTrue for success, and kFalse for failure.

In the case of of failure, lErrorList contains error reports, and $importjson has deleted a partially created output library.

In the case of success, lErrorList is empty. lWarningList may contain various warnings about the import process e.g. duplicate object idents or object names.

The error list and warning list each contain 4 columns:

- pathname: The pathname of the file containing the problem.
- errorcode: Unique integer error code for the error or warning.

- errortext: Error text corresponding to the errorcode.
- lineno: For some errors, the line number (in the file with the specified pathname) where the error occurred.

$importjson displays a working message if it executes for more than a second, and this allows the user to cancel the import, in which case $importjson returns kFalse and adds error 23433 to the error list.

4 MISCELLANEOUS NOTES

This section contains a number of miscellaneous notes and considerations related to exporting and importing the JSON library representation.

4.1 Library Dependencies

Libraries can depend on other libraries. In many cases, the presence of the external library is not required for Omnis to successfully import or export the JSON library representation. However, there are three cases that affect tokenization, and as a consequence mean the external library or libraries must be open when $exportjson or $importjson is running:

1. Design task. If the design task is in an external library, the external library must be open.
2. Superclass. If the superclass is in an external library, the external library must be open.
3. External file classes. If the code or tokenized properties use a variable in a file class in an external library, the external library must be open.

$exportjson detects the required external libraries in cases 1-3 above while it generates the JSON library representation. It adds an error to the error list when it encounters a reference to an external library that is not open, and returns kFalse. In addition, if $exportjson succeeds, it adds an array to library.json named “includes”: this is an array of all required external libraries. $importjson will fail if any of the included libraries are not open.

4.2 Tokenization

By default, Omnis tokenizes variables in external file classes using the file name and a field token. For development, I would recommend using both file and field names (to avoid untokenization issues when the external library is not open), whereas for deployment it might be more desirable for performance to use both file and field tokens.

In Studio 8.0, the only control over these tokenization options is via the browser context menu Retokenize… option. For Studio 8.1, there are some new root preferences that you can use to control this:

- $tokenizeexternalfilenames: If true,Omnis uses tokens rather than text when tokenizing external file names
- $tokenizeexternalfieldnames:If true,Omnis uses tokens rather than text when tokenizing external field names

You can use these preferences when using $importjson to control how the output library tokenizes variables in external file classes.

The values of these preferences are stored in the “defaults” entry in config.json.

4.3 Unsupported Classes

$exportjson will not export or import old plugin or iOS client remote forms.

In addition, it will not export or import #PASSWORDS or the old system table classes such as #MAWFONTS.

4.4 External Components

4.4.1 Platforms

For $exportjson and $importjson to work in a reliable cross-platform manner, the external components used by the classes being imported or exported must be available on all platforms. This does not mean they necessarily have to be fully functional, but it does mean they need to support getting and setting properties.

There are a few of our own components that need an implementation like this for various platforms e.g. Linux needs an obrowser control. These will be implemented before Studio 8.1 ships. In the meantime, $exportjson and $importjson generate errors when they encounter a missing external component.

4.4.2 Property Flags

There are some new flags which must be specified in the C++ property table of external components, before they will work with $exportjson and $importjson. Our own components already support these flags:

- EXTD_EFLAG_EXT_PROPERTIES_CRB. Set this flag for a built-in Omnis property that is stored in the external component CRB rather than the normal Omnis location. These typically correspond to properties for which ECM_BUILTIN_OVERRIDE returns qtrue - unfortunately ECM_BUILTIN_OVERRIDE requires the object to be instantiated, hence the need for this new flag.
- EXTD_EFLAG_EXT_PROPERTIES_CRB_REPORT. Like EXTD_EFLAG_EXT_PROPERTIES_CRB, except it only applies when the object is in a report class. This is messy, buy needed for one of our own components.
- EXTD_EFLAG_REPORT_MEASURE. Indicates that the fftNumber property is a report measurement. In this case, the enumStart field is actually the number of decimal places e.g. 4. You can also specify -1 in enumStart to indicate that the value is a qpridim.
- EXTD_EFLAG_ENUM_CHAR. Set this for EXTD_FLAG_ENUM properties that have a character value.

4.4.3 Property Messages

There is a new message, ECM_FMT_DEFAULTVALUE. This allows a control to return the default value for a property, in the case where it is different from the standard Omnis default (for example, zero for an integer, empty for a string).

This caters for the case where a new version of an external component adds a property, and $exportjson exports a class (containing the component) that has not been opened in design mode since the component was updated.

4.5 Special Property Values

Exported JSON can contain the following special values, which do not correspond to Omnis constants: kColorBackFill, kColorForeFill, kColorFrame.

4.6 Method Text

Testing $exportjson and $importjson has highlighted various issues with Omnis method text parsing. To ensure that a method can be rebuilt from its text, $exportjson validates each method line it exports by parsing it, and using it to re-generate the line text. If the re-generated line text does not match the original, export will fail.

In addition, the following now applies to Studio 8.1:

4.6.1 JavaScript and Text Commands

These commands now enclose their parameter in braces {} when the parameter has one or more trailing spaces, or when the parameter contains ;;. This prevents problems with whitespace stripping and inline comment detection when parsing the method text.

4.6.2 Inline Comments

If an inline comment contains the text returns (in any case) then this can also cause issues parsing method text if the command can have a Returns component. To work around this, Omnis now maps the s in the text returns in an inline comment to the Unicode character full width small or capital s.

4.6.3 Carriage Returns

I have seen libraries which contain the carriage return character inside a string in a calculation in a method. Omnis exports these characters to the method text file as Unicode 0x2ffff (guaranteed not to be a character), and restores them when it imports the method text.

4.7 Compare Classes Tool

Using the compare classes tool to compare the original library with an imported library works reasonably well. There are however some areas of which you should be aware when using this tool to compare the original library with the imported library.

4.7.1 $jshtml

I have found and fixed various issues with the generation of this property via class notation (as opposed to generating it when the design window is open).

In particular, when a control has a fieldstyle, the HTML reflected the fieldstyle rather than the control's original properties in the design window open case, but not in the class notation case. The class notation is actually correct, since the client applies the style to the control at runtime. This means that comparing $jshtml will probably fail until you have opened the class to be exported in design mode, and saved the class.

4.7.2 System Classes

Various system classes report a difference in size. I have not verified this, but I believe this to be due to a few empty CRB fields on the end of a CRB. I have not been able to see a difference in system classes which report this, when viewing them in their editor.

4.7.3 Custom Styles

I have seen one #STYLES class reporting a difference in the hascustom property. It was actually the original class that was wrong, since there were no custom styles in the affected style.

4.7.4 $comparablemethodtext

This is a new property, implemented for the compare classes tool to use when comparing method lines. It is a property of both a method and a method line, and is to be used instead of $methodtext and $text respectively by the compare classes tool.

It returns the same value as $methodtext or $text, with the exception of special case processing for comments: it strips all whitespace, maps full width small or capital s to s, and converts the comment to lower case.

Using this property avoids spurious method comparison issues caused by differences in whitespace.