The Executioner

How-To Guide

# Introduction

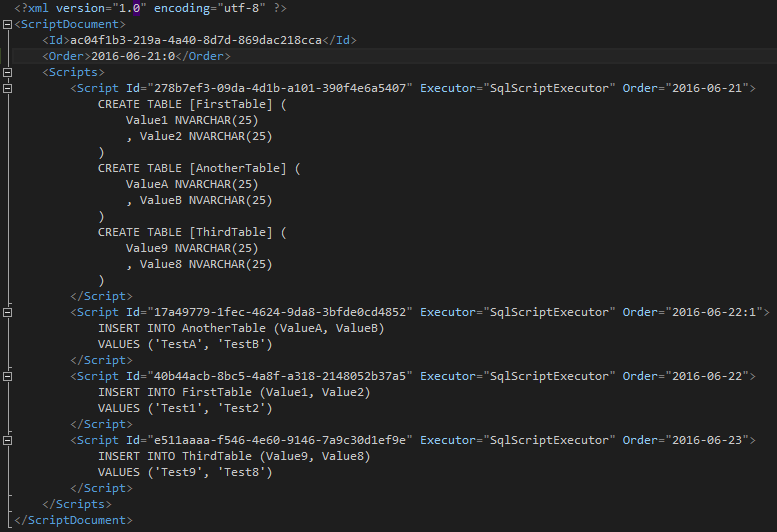
The Executioner is a library that allows for a multitude of scripts to run in requested order. This library allows the user to extend the functionality by creating their own interface implementations to provide any missing functionality they may need or want. Scripts are executed by a user creating one or many XML documents called “Script Documents”. The library can be setup to execute one of a multitude of languages and pointed to the location of where script documents can be found.

This document is meant to explain how to use the library along with how to get scripts executed.

# Script Documents

## Format

Script documents are essentially XML files that are saved with the file extensions “sdoc”. The library searches for documents with this extension in a given location and loads them in. Below is an example document.



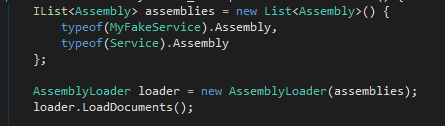
* **Id** – A GUID that ensures unique identification of a script document. This identifier is used in logging the progress of script documents that are in various states throughout the process.
* **Order** – Used to dictate the order in which a script should run. Scripts are run first by date, and then by the numeric value after the date. For example, if two script documents have the same date, then the document with the lowest value is loaded first followed by the next. Written in the format of <YYYY-mm-dd[colon]integer>
* **Scripts** – Houses a list of <Script /> tags associated with this particular script document
  + **Id** – GUID used for unique identification of each script
  + **Executor** – The name of the script executor, not fully qualified in the name, to be used when executing the script text inside of the <Script /> tag.
  + **Order** – Same use and format as the document’s usage.

# Script Loaders

Script loaders are what the library use to load the script documents. Regardless of how many documents, or scripts inside each document, have completed the script loader will load them all into memory.

## Assembly Loader

The assembly loader is used to load script documents that are saved as embedded resources in a project. This loader takes in a list of assemblies that need to be traversed for files with the extension “sdoc”. See below screenshot for example.



All script loaders implement the interface “IScriptLoader” (more information found in the “Extensibility” section). In most cases calling the LoadDocuments method is unnecessary. When this method is called, the loader will load all script documents found in the list of assemblies into the “Documents” property. Script documents are **NOT** loaded in the specified order.

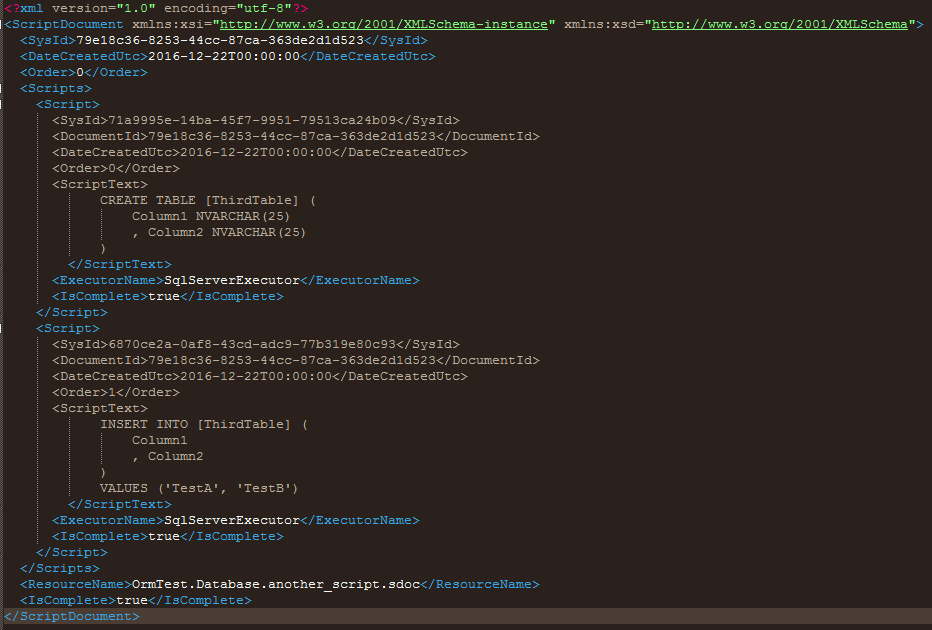
# Loggers

Loggers are used in the system to keep track of which scripts have been completed and when a document has been completed. A document is considered complete when all subsequent scripts have been executed and set to complete. The library uses the information provided by a logger to determine which scripts and script documents should be ran. Below is the list of loggers available within the library.

## File System

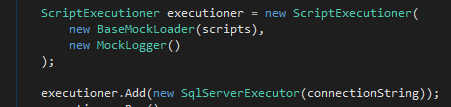
The File System logger stores all the information about the script documents in a provided location in the executing computer’s file system. When the script executioner picks up a document to start executing against, the library will create a new XML file in the provided location with the name of the file being the unique identifier found in the script document.

The XML file is in a similar format to the SDOC file but slightly different. You can see an example of a completed script document below.



# Script Executors

Executors are the part of the library that handle the execution of each script. Every script that is to be executed must have a script executor to run against. For example, if a script document contains a script that is to run T-SQL for SQL Server, then the executor for SQL Server must be available. All needed executors must be added to the API before running. The below screenshot shows an example of adding a single executor to the API. Executors can be added singularly or as a list.

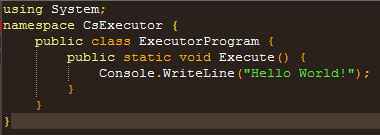


## SQL Server Executor

The SQL Server executor is used to execute T-SQL against SQL Server. The scripts run inside of a SQL transaction, so if a script fails to execute the transaction is rolled back and the process is halted. Nothing else is required to get T-SQL execution working beyond having scripts that have the Executor attribute set to ‘SqlServerExecutor’ and adding the executor to the API before running.

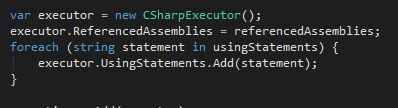
## C# Executor

The C# executor allows users to execute C# code. This executor also allows the user to reference assemblies and add using statements to provide access to other areas of code found in another DLL. The code itself found in the script nodes is wrapped around the using statements provided by the user, a namespace, a class, and a method. The below example shows what code gets generated by the executor when the script is nothing more than just a single line to show “Hello World!” to an output console.



Code found inside of a <Script /> element is wrapped inside of the Execute() method. This wrapping is done automatically (and the System using statement is added automatically as well). To add other using statements or to reference other assemblies the C# executor has properties (UsingStatements and ReferencedAssemblies) that can be used. The executor will add the appropriate using statements and reference those assemblies during the compiling process.

Each script is compiled prior to execution. Both compilation and execution can throw an exception and the rest of the process is halted with the failing script not completing. Below is an example of a C# executor that has both referenced assemblies and using statements being added.



The using statements added do not require a semi-colon or the word “using” as those will be added for you if they are missing as seen in the next screenshot.



# Script Executioner

The script executioner is the main part of the library. After creating the script documents and storing them in whichever location is desired, the script executioner requires a script loader and logger in order to be instantiated.

## Execution Result

Running the script executioner will return a result object. The result provides the following information:

|  |  |
| --- | --- |
| Property | Description |
| ScriptDocumentsCompleted [int] | Denotes how many documents have been completed |
| ScriptsCompleted [int] | Denotes how many scripts in total have been completed |

## Execution Request

In order to provide additional flexibility on how the executioner functions, a request can be sent to the “Run” method to determine what scripts you want ran. The request object that gets passed in has the following properties:

|  |  |
| --- | --- |
| Property | Description |
| ExecuteAllScripts [bool] | If false, only uncompleted scripts and documents will execute. If true, all scripts will execute regardless if they are already completed. |

## Usage

Using the script executioner is only a matter of instantiating the object and calling the “Run” method. See the screenshot below for examples and comments.

