CSV to ScriptableObject

documentation by Alessandro Iapà

Summary

1. Package Overview

1.1. Usages

As its name suggests, this package is intended for reading .csv files containing relevant data for your games to be put into the most versatile Unity's built-in data structures commonly known as ScriptableObject(s). The user's job is just to build the corresponding data structure and make sure its .csv's tables are consistent, the rest is pretty straightforward.

1.2. Design Choices

The approach used to make the package aims at a relative plug and play feature.

With this tool the user will be able to implement .csv tables containing **all of the unmanaged types** (see Microsoft Official Documentation) except for pointer(s) and user-defined struct(s), Unity's Colors, and also custom assets provided by the user like Prefabs, .png or .jpeg files such as Sprites or assets files such as ScriptableObjects etc.

Moreover, the user will be able to load **arrays** of the aforementioned data types, the only constraint is to put a **';'** character as a separator for each element in the cell when writing the database table.

Note: although Unity doesn't support (at the moment) the serialization for the **decimal** type, this tool can read it from a table and correctly load it into the corresponding asset's field, as long as the input format is correct, so accessing these type of fields is possible from code but not in the inspector of the object.

Since it is a "comma separated values" file, when writing floating point numbers, the user has to make sure not to put **','** characters as decimal points, and put a **'.'** character as decimal point instead.

This tool is intended as a .csv reader in a strict manner: the user has to provide the tables as files to the tool, so that it can be independent from external sources (not bound to a specific software or API) and it will be available for use offline.

1.3. What’s next?

Since this tool is going to be under periodic review and content update, it is possible that it will be able to not only read .csv files and load them into ScriptableObjects, but also to generate .csv files directly starting from the user generated ScriptableObjects inside the engine.

However for the moment the main focus will be on improving the tool usability and stability, focusing on making it more intuitive, user-friendly and clean, as well as maintaining the codebase and extending it according to new design ideas, necessities or issues.

Another consideration some might end up making is that it would be cool for the tool to be able to read data directly from Google Sheets tables.
Although it is a valid point, it goes out of the scope of the tool, which aims at reading .csv files no matter where they come from and especially offline.
Anyway it might be possible to have an extension or another tool like this in the future.

2. Namespace Documentation

2.1. Core

Full name: CSVReader.Core

Contains two base classes for setting up your data structures.

2.2. Attributes

Full name: CSVReader.Attributes

Contains three Attribute classes that have to be put onto fields that have to be mapped against the database or onto classes representing the corresponding .csv table.

2.3. Utilities

Full name: CSVReader.Utilities

Contains three classes whose purpose is to provide helping methods for reading .csv files and loading them into “core” classes.

2.4. Editor

Full name: CSVReader.Editor

Contains the editor script that will automatically be applied to the future database derived ScriptableObject.

3. Class Documentation

3.1. Core

3.1.1. FullDatabaseBase

Class representing the base for the complete data structure given by a set of .csv tables.
By inheriting from this class, the user will be able to define a custom data structure containing arrays of tables (new data types inheriting from TableObject).
Since this class isn't meant to have a concrete instance (ScriptableObject asset) of it, when inheriting from it the user should put the **[CreateAssetMenu]** attribute on top of it and define its specific creation path.

Main infos:

Full name	**CSVReader.Core.FullDatabaseBase**
Inherits	**ScriptableObject**
Target of	**FullDatabaseBaseEditor**

Public members:

tables	A list of all the .csv tables. Should only contain files with ".csv" extension, otherwise loading will be prevented.

Public methods:

LoadFromDatabase	Iterates over the class fields and loads the corresponding .csv files in the tables.

3.1.2. TableObject

Class representing the base for a data structure representing a roster of fields and values.

By inheriting from this class the user will be able to represent the contents of the data structure of a specific table, and since it needs to be directly mapped to a corresponding .csv file, the user must apply the **[TableIdentifier]** attribute specifying the exact name of the file, moreover, since this class is not serializable by itself, the user may provide any subclass that creates with the **[System.Serializable]** to be able to see the class displayed in the inspector along with its values.

Fields inside a derived class of TableObject should have a **[TableColumnMapper]** or **[TableColumnMapperAsset]** attribute if they have to be mapped according to the names of the columns of the table.
Fields can be either simple values or arrays of values.

Instances of the subclasses of this class should be implemented in arrays of the aforementioned subclasses as fields of a subclass of **FullDatabaseBase**.

Main infos:

Full name	**CSVReader.Core.TableObject**

Private methods:

LoadRowFields	Looks for all of the fields that have a TableColumnMapper attribute and proceeds to load them in the corresponding row.

Public constructors:

TableIdentifierAttribute	Accepts a string parameter representing the name of the table

3.2. Attributes

3.2.1. TableIdentifierAttribute

Attribute that has to be put onto classes inheriting from **TableObject** specifying the name of the .csv file it has to map its target class to. The user must be sure that the name of the .csv file is exactly identical to the name specified when applying this attribute (for example watch out for blank spaces at the end of the name of the file).

Main infos:

Full name	**CSVReader.Attributes.TableIdentifierAttribute**
Inherits	**Attribute**
Target	**Class**

Public members:

tableName	The exact name of the corresponding .csv file.

Public methods:

LoadFromDatabase	Iterates over the class fields and loads the corresponding .csv files in the tables.

3.2.2. TableColumnMapperAttribute

Main attribute that has to be put on all of the fields that needs to be mapped to the database file.

By applying this attribute to a field, the user will be able to specify a **columName** parameter that directly tells the database reader the name of the column of the .csv file it has to match to, otherwise the **name** of the field will be used.
It is particularly useful in case of code refactoring, when renaming variables while maintaining external files untouched.

Supported data types for this attribute are all of the unmanaged data types (excluding pointers and user defined structs) as well as UnityEngine.Color(s) that are represented by hex values(#FFFFFF). Moreover, it is possible to apply this attribute to array fields of the aforementioned data types.

Main infos:

Full name	**CSVReader.Attributes.TableColumnMapperAttribute**
Inherits	**Attribute**
Target	**Field**

Public constructors:

TableColumnMapperAttribute	Accepts an optional string parameter that has to be the name of the column to which the field has to be mapped.

Internal methods:

LoadField	Gets its targeting infos from its TableObject and proceeds to load either a single field or an array of fields of the same type.

3.2.3. TableColumnMapperAssetAttribute

Attribute that has to be put onto all fields that should map a project asset to a record of the database.

When applying this attribute, the user must provide a **column name** to which the field must match, and an **asset path** specifying where the asset can be found inside the project's folders.

Two more optional parameters are the **addTypePathToAssetPath** bool which adds the entire path of the class in which the object has to be referenced (useful for complex structures and namings) and the **fileExtension** string that adds the file extension that the asset must have if it is not specified in the .csv file's record.

It is possible to apply this attribute both on single values as well as on arrays of assets.

Main infos:

Full name	**CSVReader.Attributes.TableColumnMapperAssetAttribute**
Inherits	**TableColumnMapperAttribute**
Target	**Field**

Public constructors:

TableColumnMapperAssetAttribute	Needs two strings, first one is the column name, the second one is the path at which the asset is located. Accepts two optional parameters, a bool saying that the class path referencing the object has to be added to its complete path, and a string representing the asset file extension.

Internal methods:

LoadField	Gets its targeting infos from its TableObject and proceeds to load either a single field or an array of fields of the same type.

3.3. Utilities

3.3.1. <u>IOMediator</u>

Simple object that manages the .csv file reading and conversion into an Object(s) representation of the table, with each record separated by a comma (**‘,’**).

Main infos:

Full name	**CSVReader.Utilities.IOMediator**

Public methods:

GetTableData	Given the table Object, returns an IList<IList<Object>>, a matrix of Object(s), each one representing a record of the database table.

3.3.2. <u>TableDataReaderUtility</u>

Object holding essential data and methods to start the process of reading each value of the database’s cells.
Each instance of this class is used to temporarily store useful infos such as the table name, an Object representation (provided by the **IOMediator**) of the .csv file, a dictionary of all the columns of a table and the index associated to them and so on.

Moreover, this class provides a method to read a single record of the table using the column name and the index of the row at which the value is located and converting it into a meaningful string for the **TableLoaderUtility**.

Main infos:

Full name	**CSVReader.Utilities.TableDataReaderUtility**

Private members:

tableTitle	A string containing the title of the table.
columnMap	Dictionary containing column name string - column index couples.
columNames	A list of strings containing all the columns of a table in order.
values	An IList<IList<Object>> or a matrix of Object(s) containing the values of each record.

Public properties:

RowsCount	Returns the amount of rows of values present in the table, excluding the first row which contains the column names.
ColumnsCount	Returns the amount of columns of the table.
TableTitle	Returns the name of the table this object refers to.
TableRawData	Returns an IList<IList<Object>> or a matrix of Object(s) containing the values of each record.

Public constructors:

TableDataReaderUtility	Needs the table’s name and the matrix of Object(s) representing the table’s content.

Private methods:

ReadRecord	Given the column name and the index of the row, gets all the values of the row, then uses the name of the column to find the index at which the object value is located, and returns it.

Internal methods:

GetRecordStringAt	Returns the string value representation of the object value returned from the ReadRecord method.
HasColumn	Checks whether the given column name is present in the dictionary of column names.
GetColumnName	Returns the name of the column that corresponds to the index parameter.

3.3.3. <u>TableLoaderUtility</u>

Object responsible for handling all of the data conversions from pure strings into the specific target types and values, as well as providing the capability of bridging between data management classes and data representation classes.

This class also provides some more utilities to run security checks before proceeding with the actual data loading and values' setting.

Main infos:

Full name	**CSVReader.Utilities.TableLoaderUtility**

Private members:

tableReader	A reference to a **TableDataReaderUtility** holding the data read from the .csv file.

Public properties:

tableTitle	The name of the table referenced by the **tableReader** object.
TableLenght	Returns the amount of rows of the table referenced by the **tableReader** object.

Private constructors:

TableLoaderUtility	Needs a **TableDataReaderUtility** to handle and manipulate the data representation of the table.

Private methods:

ProcessFieldAndLoad	Given the infos of the field, filters between the available supported data types to determine which parsing should be performed.

Internal methods:

LoadField	Bridges between the object on which to apply data modifications and the actual fields of the database.
GetCharValue	Returns the parsed value of the record under the form of a char.
GetBoolValue	Returns the parsed value of the record under the form of a bool.
GetFloatValue	Returns the parsed value of the record under the form of a float.
GetDoubleValue	Returns the parsed value of the record under the form of a double.
GetDecimalValue	Returns the parsed value of the record under the form of a decimal.
GetSignedByteValue	Returns the parsed value of the record under the form of a sbyte.
GetShortValue	Returns the parsed value of the record under the form of a short.
GetLongValue	Returns the parsed value of the record under the form of a long.
GetUnsignedByteValue	Returns the parsed value of the record under the form of a byte.
GetUnsignedShortValue	Returns the parsed value of the record under the form of a ushort.
GetUnsignedIntValue	Returns the parsed value of the record under the form of an uint.
GetUnsignedLongValue	Returns the parsed value of the record under the form of a ulong.
GetColorValue	Returns the parsed value of the record under the form of a UnityEngine.Color.

HasColumn	Returns true if the given column string exists.
GetColumnName	Returns the processed name of a column if a postfix is applied.
GetStringAt	Returns the string representation of an object value in the record specified by the columnName and row index parameters.
CreateField	Returns the parsed object that matches the fieldType and the fieldValue parameters.

3.4. Editor

3.4.1. FullDatabaseBaseEditor

Basic custom inspector for all of the objects deriving from **FullDatabaseBase** that adds a loading button when all of the security conditions are fulfilled.

Main infos:

Full name	**CSVReader.Editor.FullDatabaseBaseEditor**
Inherits	**UnityEngine.Editor**

Overrides:

OnInspectorGUI	Here the loading button is displayed after some security checks.

Protected methods:

LoadData	Uses utilities for reading all of the tables and then asks the database to read and fill itself.

Private methods:

GetAllData	Generates all of the data reading utilities and returns them as a List used to iterate over the tables of the database.

4. Examples

4.1. The test folder

Inside this package, there is a “Test” folder containing some example assets that show how this tool is intended to be used, as long as two demo .csv tables with two different record layouts, one containing simple values, while the other contains arrays of values.

4.2. Tests

Main folder containing: three different prefabs referenced by the two database classes; the other folders of the examples’ section.

4.3. Classes

Contains three classes:

- **TestSingleValuesDatabase** which is a container class that inherits from **TableObject** mapping what is contained in the corresponding file pointed out by the **[TableIdentifier]** attribute on top of it.

Each field of the table is either tagged with the **[TableColumnMapper]** attribute or the **[TableColumnMapperAsset]** attribute.
For each one of the unmanaged data types fields, the attribute uses the *optional* parameter containing the name of the column in the database, otherwise the name of the field will be used, while for the asset attributes (in this specific case they are GameObject(s)) the name of the asset is a mandatory parameter that needs to be applied, as well as the path at which the asset can be found.
Since it might a choice driven by what is more practical when dealing with a .csv table, it is possible to specify the extension of the aforementioned asset, in the case of this folder, all of the assets are mapped in the attribute with the .prefab extension, but it is possible to map other assets with other file extension. By not puting the extension, it is implicit that it will be provided inside the database record.

- **TestArrayValuesDatabase** which is analogous to the previous class, except for the fact that each field is an array of the corresponding type.
The rules for mapping values and assets are exactly the same as before, it is required, however, to separate values in each record using a **‘;’** character.

It is important to make sure that there are no characters such as ‘/’ and ‘\’ since they will be removed, so the user has to make sure that when entering data none of those characters is present (pay attention when pressing “enter”).

- **<u>TestFullDatabase</u>** is the inheritor of the **FullDatabaseBase** class and it represents the entirety of the database (i.e. the collection of tables).

Each table is represented by an array of inheritors of **TableObject**, like in the case of this class, which contains two arrays: a **TestSingleValuesDatabase** one and a **TestArrayValuesDatabase** one.

Since the class is the implementation of a **ScriptableObject** the user must provide it with the **[CreateAssetMenu]** attribute in order to create concrete implementations of the databases, by filling the list of necessary .csv tables and pressing the "Load" button.

4.4. <u>Data</u>

This folder contains the ScriptableObject instance created by the **TestFullDatabase** class, named **Test Database.**

The asset contains a list with two .csv files, that are the tables that have to be loaded. Their name must be the same as the parameters specified above the **TableObject**(s) implementations.

Loading the database will be prevented if one or more tables' values in the list is empty, or if there are no lists, or if the user tries to put a file with the wrong extension.

4.5. <u>Files</u>

This folder simply contains the two .csv tables named exactly as the names specified in the **[TableIdentifier]** attributes above the **TableObject** implementations.

5. Troubleshooting

5.1. Loads of warnings

One of the most common causes of warnings generated by the tool happens when trying to load empty asset fields, there is nothing to worry about, but if this warning is present even when all of the corresponding assets' records have a value, there might be a problem during importing or an user generated error when manually filling the .csv file, it is in fact common to press enter when editing the table, and this might cause the insertion of an invisible '\' or '/' in the record, especially when editing arrays.

5.1.1. Other warnings

It is less likely to happen that a database field does not inherit from **TableObject**, but since it might happen, it is important to notice that this tool loads only objects that inherits from **TableObject** or from its inheritors, so any other object will not be considered as part of the database and has to be loaded manually.

When trying to read a record value mapped to a *char* field, it is possible to encounter a warning if the string of the record has more than one character. In this case the execution won't stop and only the first character of the string will be considered meaningful and loaded into the field.

5.2. Errors

Since the tool works with files and strings, it is really common that a simple typo throws a bunch of errors and exceptions.

It is important to point out that this tool won't try to modify the user's generated .csv files, so no exceptions or errors are caused by the tool corrupting the integrity of those documents, moreover the tool won't try to figure out where the error was and fix it by arbitrarily and openly modifying the file, since it might be a choice of the user to have a file's record set up that way.

5.2.1. Attributes' errors

Attributes will throw exceptions or errors when type conversion is impossible, so it is a good habit to check the correspondence between a field and the value mapped in the record of the database (i.e. do not put a string saying "hello" in a cell where an int is expected).

Specifically for the Assets' case, errors are generated when the given asset is not found in the specified directory, so it is important to check both the validity of the path and the name of the asset.

Note: an exception might (rarely) be raised for different reasons, such as trying to access a column name or row index that doesn't exist.

5.2.2. <u>**Core classes' errors**</u>

One of the most common errors is logged whenever a Table is not found, this might happen when the name of the actual .csv file does not match exactly with the name specified by the **[TableIdentifier]** attribute's tableName parameter. Sometimes white spaces at the beginning or at the end of the file name might cause this error.

Although it should never happen, this tool might raise an exception when trying to load a new instance of a table to read its rows and columns, depending on the type of exception that was raised, it is possible to figure out which was the cause of the exception, but it is always a good habit to check all the file names and attributes' names.

5.2.3. <u>**Utilities classes' errors**</u>

It is possible that when reading a .csv file an exception is raised, it will be logged and the entire code block will return a null value.
The user should make sure that files are intact and uncorrupted.

Again, when trying to load the database and creating the necessary utilities' instances in the process, it is possible that exceptions are thrown if the matrix of values representing the table is either null or empty or, again, if a column name is already present in the document.

When reading a row, it is possible to encounter exceptions when trying to look for a column name that doesn't exist, so make sure that the names of columns in the document match with the names specified on fields' attributes or fields' names match.
Another column related error might happen when trying to find and access a column by its index: in fact, if the index is not in the range of the list of column names, an error will be logged.

It is possible to encounter an exception when reading a row in two cases: the first and most plausible one is that when a row resulted to be null, in this case, it means that there might be a "ghost" row in the document; for the same reason, when trying to access a row by its index, it will result null or non existent if the index is out of the range of the collection of rows.
Finally, when reading proper record values, it is possible to encounter

the most common types of errors, those ones are mainly related to data conversion and parsing of values.

When loading a field, an exception might be raised if either the row index resulted null or out of range, or if the instance of the **TableObject** resulted null for some reason (99.9% of the time this exception won't be raised).

It is more common to see logged an error that says that the field cannot be loaded because it is not an unmanaged data type, this happens when trying to mark a field which is not unmanaged (such as a struct or a class) with the **TableColumnMapper]** attribute, therefore it is up to the user to make sure that attributes are placed correctly.

When parsing a numeric field errors will be logged if the targeted field is an "unsigned" data type and the input value coming from the record has a negative sign preceding it.
Each time this error is logged, the execution is not stopped, but the returned value will be 0, independently of the numeric type.

Final consideration and suggestion:

Most of the errors that came up during testing and by keeping on using this tool extensively during development ended up being 99.9% of the times caused by human error.
Although this tool tries to catch and log errors and exceptions of any kind, in order to provide the user a meaningful debugging system, it is important to always run a double check both on names of fields or on names that were put on attributes.

6. About the author

I’m Alessandro, I live in Milan (Italy), I’ve studied Game Programming and I’ve started working in the game industry as a Unity Game Programmer both as part of a team and as external consultant for other companies.

The experience taught me the importance of speed and flexibility in development, and thanks to the possibilities that Unity offers in terms of expandability and customization I’ve started working on some tools in my spare time.
Those tools come from necessities encountered during the development of various games, and I’ve found them essential for improving the everyday experience, so I’ve decided to start rebuilding them for a consumer’s use.

If you are using one of my tools, do not hesitate to contact me in case of need or if you have any suggestions, I’m always open for any new ideas that I can develop for others that found to be useful.

Consider taking a look at my portfolio here (might be a little outdated sometimes but I try to pay attention to it as soon as I can).

If you need help about some of my tools that you are using or if you need help or confrontations of any kind feel free to email me at: alessandro.iapo99@gmail.com

I hope you find my tools useful and I wish you a good job, best of luck devs!