



## XML Mirror User's Guide

Copyright © 2012-2017 Data Juggler Software

### Background

XML Mirror was first created because I used to work at a job where my duties involved consuming large amounts of data via web services from popular enterprise social networks such as [Yammer](#) & [Tibbr](#). I have never been a fan of parsing xml by hand as the amount of code required to handle numerous types of nodes in the xml is very tedious and time consuming.

XML Mirror was created using the same C# Class Writer as is used in RAD Studio Code Generation Toolkit, available at <http://radstudio.codeplex.com> (version 6.0 should be released by the time you read this or soon thereafter).

I tried to sell Xml Mirror for a few years, but I am horrible at marketing because I spend all my time writing code. I made the decision to release this open source because I feel Xml Mirror is too useful to reside only on my pc.

### Parsers

The first version of Xml Mirror only created parsers. XML Mirror Parsers use partial classes in the same manner as they are used in RAD Studio Code Generation Toolkit. The base class ([\[MirrorName\].base.cs](#)) is rebuilt every time you rebuild your parser, but the custom class is only created if it does not exist. This allows you to update your base class if you discover new fields you need to parse (which happens to me often) and any code you

have created in the custom class ([\[MirrorName\].custom.cs](#)) will not be overwritten.

## Writers

Writers are new to version 2.0 and have been on my to do list for many years, but I finally managed to find the time. I have used the Xml Serializer from Microsoft and it works sometimes, but I have experienced problems using it. I have also written Xml by hand more times than I care to admit. Xml Mirror uses reflection to create C# Xml Writers that export properties for the selected object.

## Parsers Overview

There are two modes for using XML Mirror; you can parse a single instance of an object (aka Singleton), or you can parse a collection ([List<T>](#)) of objects.

The following code snippet shows how easy it is to parse a collection of messages (Message objects) from the source xml text:

```
// get the sampleMessageText
string sampleMessageText = GetSampleMessageText();

// if the sampleMessageText exists
if (TextHelper.Exists(sampleMessageText))
{
    // create an instance of the messageParser
    MessageParser parser = new MessageParser();

    // parse the messages
    messages = parser.LoadMessages(sampleMessageText);

    // Display the messages
    this.DisplayMessages(messages);
}
```

## Documentation

You are reading this so you must have found the 'Help' menu to launch this file. I have done my best to explain XML Mirror in this document, but I also understand I wrote this software, so things that are obvious to me might not be to someone who isn't as familiar. If you feel there is anything I have left out or needs to be explained in more detail, ask any questions on the Azure DevOps home at <https://dev.azure.com/datajuggler/>

## YouTube Video

Click the YouTube link in Xml Mirror to view the Xml Mirror 2.0 video.

## Parsing Basics

The parsing works the same whether you are parsing a collection of objects or a single instance of an object. The entire xml document is loaded with the following code snippet:

```
// create an instance of the parser
XmlParser parser = new XmlParser();

// Create the XmlDocument
this.XmlDoc = parser.ParseXmlDocument(userXmlText);

// If the XmlDocument exists and has a root node.
if ((this.HasXmlDoc) && (this.XmlDoc.HasRootNode))
{
    // parsing implementation goes here
}
```

## Parsing Single Items

As mentioned previously there are two types of Mirrors you can create, single instance parsing and you can parse a collection (List<T>) of objects. The screen shot below demonstrates a 'Single User Parser', which parses a single instance of a 'User' object.

The following code snippet shows how to load a 'User' object from the source xml text:

```
// set the documentText
string sampleUserText = GetSampleUserText();

// if the sampleUsersText exists
if (TextHelper.Exists(sampleUserText))
{
    // create an instance of the userParser
    SingleUserParser parser = new SingleUserParser();

    // parse the users
    singleUser = parser.ParseUser(sampleUserText);

    // Display the user
    this.DisplayUser(singleUser);
}
```

## Parsing Collections (List<T>)

Parsing collections of objects works the same as parsing a single instance except that a generic collection is created when the Collections Action Node is encountered.

We will discuss how to create parsers for collections in the section '**Creating Mirrors**', but basically you assign an [Action Node](#) (explained later) as being the 'Create Collection Node' and another as being the 'Create Singleton Node'.

As the nodes in your source xml are iterated, when the 'Create Collections Node' is encountered your collection is instantiated:

```
// get the full name for this node
string fullName = xmlNode.GetFullName();

// if this is the new collection line
if (fullName == "response.messages")
{
    // Raise event Parsing is starting.
    cancel = Parsing(xmlNode);
}
```

## Parsing Events

There are two events that are fired during the parsing process of a single object that help you perform customizations that extend the code generated by XML Mirror.

### Parsing (XmlNode node, ref [Object Type] object)

The 'Parsing' event is fired **before** the parsing of the object begins. This is useful for setting properties that may not be in the source xml.

This is also useful for validation purposes. To cancel the parsing of an object, set Cancel to true in the **Parsing** event and the 'Parse' method will return null.

## **Parsed (XmlNode node, ref [Object Type] object)**

The 'Parsed' event is fired **after** the parsing has completed. Just as with the Parsing event, this allows you to set any properties that may need to be set after the parsing has completed.

This is another place you may validate your object to ensure all required values are set. To cancel the parsing of an object, set Cancel to true in the parsed event and the 'Parse' method will return null.

**Note:** Xml Mirror creates the event for 'Parsing' for collections even though the create collection node was not set. The reason the event for collections is created in 'Singleton' parsing mode is that the [Parser Name].custom.cs class is only created if it does not exist. If you ever modify your mirror and add a 'Collection' node to the mirror it would not be possible to update the custom partial class to include the 'Parsing' event for collections.

**You are free to delete the Parsing collection event if you desire.**

## **Collection Parsing Events**

### **Parsing (XmlNode node)**

In addition to the two events mentioned previously in the **Parsing Single Objects** section, another 'Parsing' event is raised before the collection node is encountered.

This is useful for setting any lookup values or to perform validation on the entire parsing batch. Set 'Cancel' to true and the parsing batch will be cancelled and null will be returned.


## **Creating Mirrors**

A Mirror contains all the information needed to parse the source xml and map the associated field links to properties in your 'Target Class'.

XML Mirror launches in 'New Mirror' mode so you may notice the lack of a 'New Mirror' button.

## Steps to create a Parser

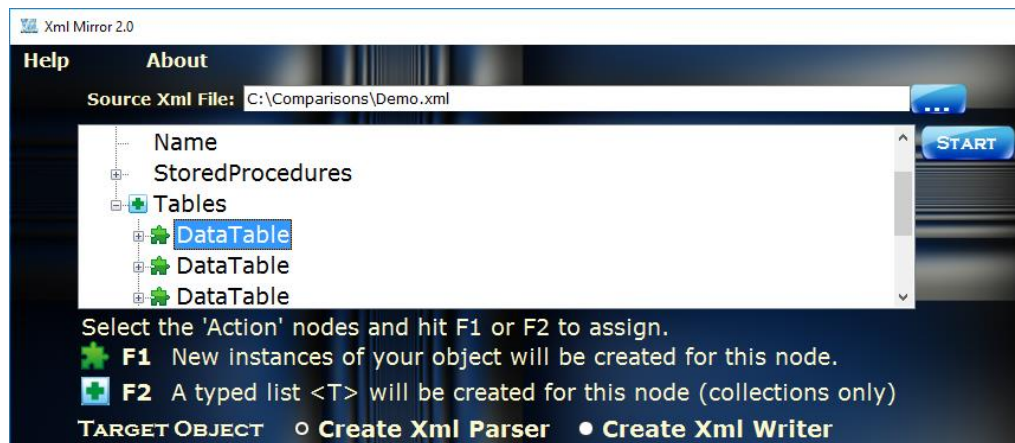
### Select the Source XML

Enter the path to the source xml file or click the ellipses button  to browse for the source xml file.

**Note:** If for any reason your source xml does not have a root node, you must either modify the Xml source or you can “create” a root node so that XML Mirror can parse the document.

```
// surround the sourceXml with a 'Node'  
sourceXml = "<InstanceNode>" + sourceXml + "</InstanceNode>";
```

Click the 'Start' button to parse the source xml file:



### Action Nodes

The action nodes are the nodes in your source xml that your parser will create either a collection of objects (F2) or a new Instance of an object (F1) when they are encountered.

**Note:** You must select an instance node, but a collection node is optional.

### Collection Nodes (F2)

If you are creating a collection, you should create the collection mode first by selecting the collection node and hitting F2. There can be only one (sounds like a Highlander movie) collection node in your Mirror. If you assign a node as being a Collection Node any other nodes that are currently assigned as being a collection node will be removed.

Whenever a node matching the text of the mirror's 'Collection Node' is reached, the parser will instantiate a new collection (List<T>) of objects of the type of object as the 'Target Class' (explained later).

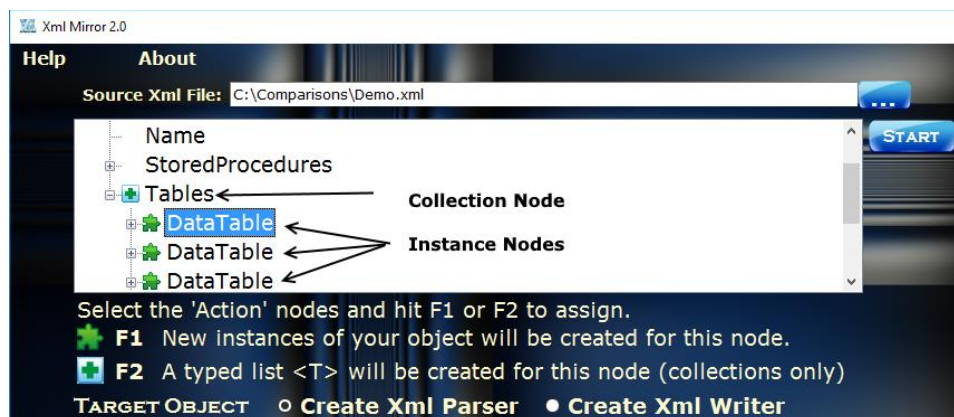
```
// get the full name for this node
string fullName = xmlNode.GetFullName();

// if this is the new collection line
if (fullName == "response.messages")
{
    // Raise event Parsing is starting.
    cancel = Parsing(xmlNode);

    // If not cancelled
    if (!cancel)
    {
        // create the return collection
        messages = new List<Message>();
    }
}
```

## Instance Nodes (F1)

Instance Nodes behave differently than Collection Nodes in that Instance Nodes are assigned to all matching nodes as shown in the picture below:



When you parse an object using a parser created by Xml Mirror, every time the 'Instance Node' is encountered a new instance of your object will be created

## Selecting the Target Object

**TARGET OBJECT** • **Create Xml Parser** ○ **Create Xml Writer**

Target DLL: C:\Projects\XmlMirror\bin\Debug\XmlMirror.Runtime.dll

Target Class: XmlMirror.Runtime.Objects.Mirror

Writer Name: MirrorsWriter

Namespace: XmlMirror.Xml.Writers

Output Folder: C:\Projects\XmlMirror\Xml\Writers

Class XmlMirror.Runtime.Objects.Mirror

- ClassName
- CollectionNodeName
- FieldLinks
- Fields
- MirrorType

**VIEW**

Your target object can be any .NET object provided it does not require any dependencies to be instantiated. If you encounter any errors attempting to reflect your object then you will need to create a subset of your object in a separate class library project (DLL).

**Note:** The 'Target Class' must have a default constructor (meaning a constructor with no parameters) or XML Mirror will not be able to create new instances of the target object. If you need to set any additional properties or initializations to your objects, use the 'Parsing' or 'Parsed' events.

I had to restructure Xml Mirror because originally Xml.Mirror.Runtime contained a reference to System.Drawing and System.Windows.Forms.

## Parser Name or Writer Name

The 'Parser Name' or 'Writer name' is created as the name of the target object plus Parser or Writer. This is also the class name of your object and the filename for the C# output. The name comes from the 'Target Object' name, but you are free to change it if you desire before you build.

## Namespace:

The namespace must be set so that the output files are created for your parser project.



## Output Folder:

The output folder must exist and the current 'Windows' user must have write access to this folder. As explained earlier each mirror will create two partial classes in the output folder:

[MirrorName].base.cs                      [MirrorName].custom.cs

The base class is recreated (overwritten) every time you build your parser, while the custom class is only created if it does not exist.

## Creating Field Links (Parsers Only)

A field link maps the relationship between an xml node and a property in your target object. To create a field link, drag an xml node from the source xml and drop it on the desired property in the target object.

## Auto Fill

New to version 2.0, I created an 'Auto Fill' button that will attempt to automatically map Xml Nodes to the matching property of the same name.

The best part about field links is that XML Mirror will detect the data type of the property and when the parser is created the corresponding method in DataJuggler.Core.UltimateHelper will be called:

```
case "response.id":  
  
    // Set the value for user.UserID  
    user.UserID = NumericHelper.ParseInteger(xmlNode.FormattedNodeValue, 0, -1);
```

In the 'ParseInteger' method above, 0 is the default value and -1 is the value returned if an error is encountered parsing the string value as an integer. You are free to change the values but since this code is created in the base class this would be overwritten if you rebuild your project using XML Mirror. A better solution is to change the value in the 'Parsed' event, explained later in this document.

The helper classes all come from one of Data Juggler's open source projects:

**Data Juggler Ultimate Helper** <http://ultimatehelper.codeplex.com>

**Note:** Xml Mirror includes a more current version of the Ultimate Helper project. I will update the Ultimate Helper project soon.

## XmlMirror Runtime

Originally XmlMirror.Runtime.Util contained a copy of many of the classes in Ultimate Helper. I had two reasons for removing the helper classes from XmlMirror.Runtime.

1. It was difficult for me to keep two projects updated when classes in Ultimate Helper are updated.
2. When I created the writing features of Xml Mirror, I threw out the manual way I had been saving Xml Mirrors and replaced them with an Xml Writer created using Xml Mirror. When I tried reflecting XmlMirror.Runtime, I received an error because the project contained a reference to System.Windows.Forms and System.Drawing.

## Sample Projects

In the early versions of Xml Mirror I included a very basic sample project that included a couple of parsers. Now that Xml Mirror is used to build itself, a sample project is not necessary. You can view the following directories of the source code to view the parsers and writers built using Xml Mirror:

[Source Code Directory]\XmlMirror\Xml\Parsers

[Source Code Directory]\XmlMirror\Xml\Writers

## DB Compare 2.0

<https://dbcompare.codeplex.com>



The driving factor which made me decide to create the Writers for Xml Mirror was I needed an easy way to export objects for DB Compare.

At my current day job, I must keep our dev, test and production servers synchronized with the same schema as my local dev machine. Prior to having this tool, whenever I performed an update I had to script my entire dev database and create a temporary database to compare the schemas.

Now I export my local dev SQL Server to Xml and perform a compare and it saves me lots of time on each server.

## Supported Data Types

String (default value for any type that cannot be determined)

Int

Double

DateTime

GUID

## Enums & Enum Helper (Parsers Only)

New to version 2.0, I created an Enum Helper to parse enumerations. The only thing I require you to do is to select the default enum value.

The code created to parse an enum is written commented out:

```
// Set the value for dataTable.Scope  
// dataTable.Scope = EnumHelper.GetEnumValue<DataManager.Scope>  
(xmlNode.FormattedNodeValue, DataManager);
```

Uncomment out the line and select the default value for the enum.

## Custom Data Types

There are numerous ways to parse data types that are not natively supported by XML Mirror. The Parsed event is the most logical place for parsing custom objects.

## Parsed Event

In addition to validation, another useful feature of the 'Parsed' event is to perform operations that must be performed after the object has been parsed.

This example is from the MirrorsParser.custom.cs, and demonstrates how to load objects that are not natively parsed.

```
public bool Parsed(XmlNode xmlNode, ref Mirror mirror)
{
    // initial value
    bool cancel = false;

    // if the mirror exists
    if (NullHelper.Exists(mirror))
    {
        // Create a new instance of a 'FieldLinksParser' object.
        FieldLinksParser fieldLinksParser = new FieldLinksParser();

        // Parse the FieldLinks (if present)
        mirror.FieldLinks = fieldLinksParser.ParseFieldLinks(xmlNode);

        // Create a new instance of a 'FieldsParser' object.
        FieldsParser fieldsParser = new FieldsParser();

        // parse the FieldValuePairs (if present)
        mirror.Fields = fieldsParser.ParseFieldValuePairs(xmlNode);
    }

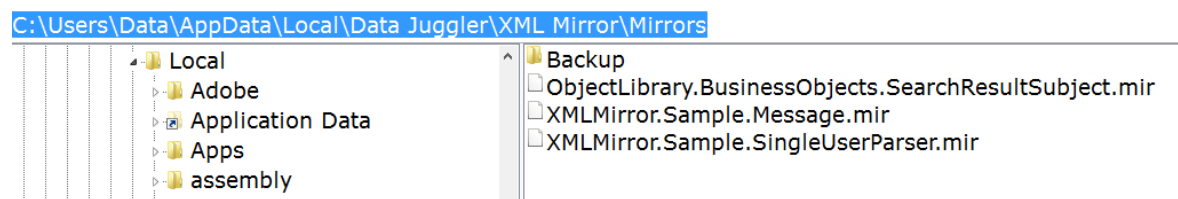
    // return value
    return cancel;
}
```

## Saving Mirrors

To save a mirror click the 'Save Mirror' button, this will launch the save dialogue:



Mirrors are saved in the following folder, where the name of the logged in user is 'Data':



## Mirror Backups

The first version of XML Mirror relied on XML Serialization to save the mirrors. This worked most of the time, but occasionally the serialization would corrupt the file and a mirror would be lost.

I was frustrated on losing mirrors that were mapped, so I created a backup folder for the mirrors and started saving a backup copy of the mirror every time save is called.

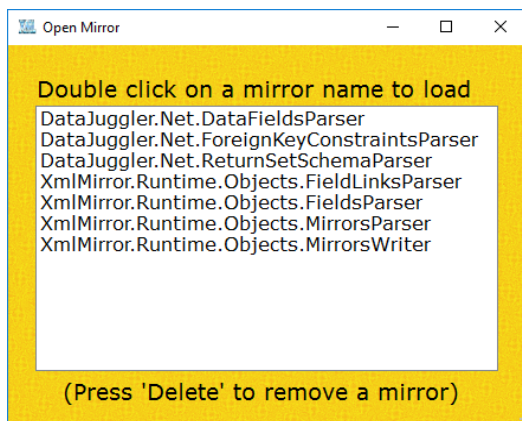
The backup file name is appended with a timestamp to keep the filename unique in the backup's folder.

I have since changed the way mirrors are loaded and saved. Now I load and save mirrors using Parsers and Writers created by Xml Mirror itself.

I left the backup code in just in case a file ever becomes corrupted or to give you a way to revert to an earlier version of a mirror if needed.

## Loading Mirrors

To load a mirror, click the 'Load Mirrors' button and select the mirror from the list given:



Loading a mirror will load all properties and field links (for parsers), provided the xml node exists in the parsed xml document and the property still exists in the target object.

## Building Parsers and Writers

After you create or load a mirror, you are ready to build a parser and / or a writer. Click the 'Build Parser' or 'Build Writer' button and XML Mirror will create your parser or writer respectively.

### Parser Output

The parser will create the base class every time you build.

The custom class will only be created if it does not already exist; this allows you to place any custom methods or properties that will not be overwritten if you rebuild your parser using XML Mirror.

### Writer Output

Writers only create one single file, and I have yet to find a reason to use partial classes for writers.

When a Writer is written, any child properties that are of type List<T> (System.Generic.Collections) will require you to create a writer for this object as shown here:

```
// Write out the value for Fields

// Create the FieldsWriter
FieldsWriter fieldsWriter = new FieldsWriter();

// Export the Fields collection to xml
string fieldValuePairXml = fieldsWriter.ExportList(mirror.Fields, indent + 2);
sb.Append(fieldValuePairXml);
sb.Append(Environment.NewLine);
```

## Visual Studio Setup

You will need a reference to the following two projects:

**XMLMirror.Runtime.Dll**  
**DataJuggler.Core.UltimateHelper**

At one point, I created NuGet packages for both of the above projects, but keeping NuGet updated and working for all versions of the Dot Net Framework takes more time than I have available.

You are welcome to move or combine any of the code in the above projects in any way you desire. I recommend keeping them separate since replacing a project is simple if an update is published to either project.

## Including Project Files

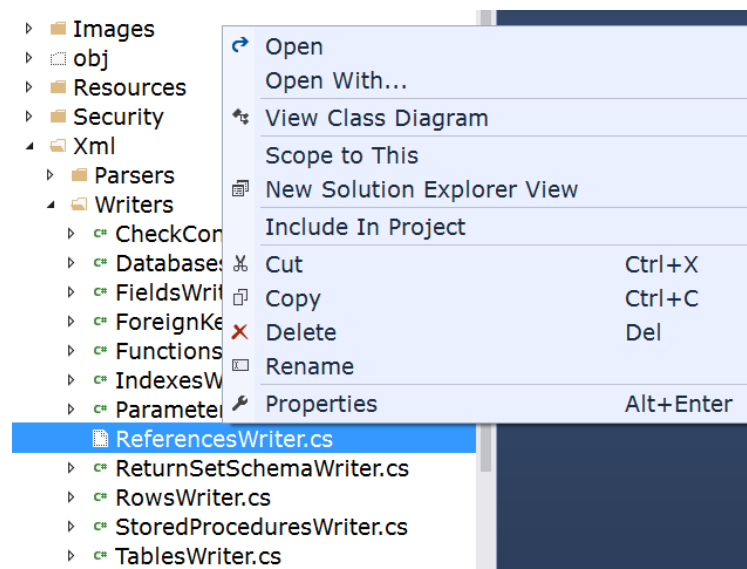
### Parsers

After you have built your parser(s), browse to the 'Output Folder' and include the following two files in your project:

[Mirror Name].base.cs  
[Mirror Name].custom.cs

### Writers

Writers only create one file as shown in the picture below:



If you feel that creating partial classes for writers has value, please create a topic on the discussions tab of the Code Plex project at:

<https://xmlmirror.codeplex.com>

## **Deployment**

### **XMLMirror.Runtime.dll**

### **DataJuggler.Core.UltimateHelper**

Both classes above will be compiled into the bin folder of your project. If you publish a web project or build an installer for your application, ensure the above two objects are included when you deploy.

### **XmlMirror.Runtime**

This class loads the Xml Documents that all parsers use at runtime to load objects and collections.

You can find this file in the project directory:

[XmlMirror Project Folder]\Runtime

### **DataJuggler.Core.UltimateHelper**

This project contains multiple helper classes used by Xml Mirror.

Xml Mirror ships targeting the Dot Net framework 4.61; you are welcome to change this to any version of the Dot Net Framework 4.0 or above.

## **FAQ**

### **Does XML Mirror work with Visual Basic.Net?**

The short answer is yes, but I have never done it.

Here are the steps required to use XML Mirror in a VB.Net project.

Create a C# class library project for the writers and parsers output by Xml Mirror. Reference the C# parser library from your Visual Basic project.

Or

Learn C#! (Recommended)

I was a Visual Basic 3.0 – 6.0 programmer for years before learning C# and haven't missed VB at all. The main reason the company I worked for switched from VB to C# was that my boss at the time felt C# programmers make more money and are more in demand.



## **Does XML Mirror work with Visual Web Developer?**

I am sorry but I cannot answer that question as I have never used Visual Web Developer. If Visual Web Developer allows you to create and reference class libraries than the answer should be yes. If anyone out there has used Visual Web Developer and would like to answer this question it would be appreciated. Please email me if you can provide some insight into Visual Web Developer.

## **Support**

Xml Mirror is provided free of charge, so I cannot support the project for free. I will try and answer all issues and discussions on the Code Plex site at

<https://xmlmirror.codeplex.com> .

Have a large XML project? I can build custom parsers for you, contact me for a free quote.

## **Miscellaneous**

### **Regionizer**

<http://regionizer.codeplex.com>

There were some breaking changes to Visual Studio 2017 extensibility. I have already created a working version for me using VS 2017 RC, but I have had a few problems I need to debug before I publish this release.

Check the above project later this month (January 2017) as I hope to publish an update as soon as I can.

### **Regionizer Intro Movie**

[https://www.youtube.com/watch?v=dtHtVAT\\_xW0](https://www.youtube.com/watch?v=dtHtVAT_xW0)

If you have viewed any source code that was created by Data Juggler, you will soon realize that I am a regionaholic. Regionizer is a free open source Visual Studio package that formats your C# documents into regions that are sorted alphabetically by the type of code: Events, Methods, Properties, etc.

Now that I have used Regionizer for about 7 years, any code that is not formatted into regions looks like spaghetti code to me. I cannot program without regions I have gotten so use to them.

Please download Regionizer for yourself.

## **Ratings and Reviews**

I hate to sound critical of open source software consumers, but I feel most take the "free" code and software for granted.

I donate hundreds of hours of my time every year to creating open source projects. A few minutes of your time to leave a review or a rating is a fair price to pay for the quality of the software you are receiving free of charge.

Thank you for leaving a rating and / or a review if you like this project.