# Background

## 1.XML

XML is a markup language created by the World Wide Web Consortium (W3C) to define a syntax for encoding documents that both humans and machines could read. It does this using tags that define the structure of the document, as well as how the document should be stored and transported.

It is probably easiest to compare it to another markup language with which you might be familiar—the Hypertext Markup Language (HTML) used to encode web pages. HTML uses a pre-defined set of markup symbols (short codes) that describe the format of content on a web page.

The thing that differentiates XML, though, is that it is extensible. XML does not have a predefined markup language, like HTML does. Instead, XML allows users to create their own markup symbols to describe content, making an unlimited and self-defining symbol set.

Essentially, HTML is a language that focuses on the presentation of content, while XML is a dedicated data-description language used to store data.

XML is often used as the basis for other document formats—hundreds, in fact. Here are a few you might recognize:

- RSS and ATOM both describe how reader apps handle web feeds.
- Microsoft.net uses XML for its configuration files.
- Microsoft Office 2007 and later use XML as the basis for document structure. That is what the “X” means in the .DOCX Word document format, for example, and it is also used in Excel (XLSX files) and PowerPoint (PPTX files).

So, if you have an XML file, that does not necessarily tell you what app it is intended for use with. And typically, you will not need to worry about it, unless you are the one designing the XML files.

## 2.JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with a diverse range of functionality in data interchange including communication of web applications with servers.

JSON is a language-independent data format. It was derived from JavaScript, but many modern programming languages include code to generate and parse JSON-format data. JSON filenames use the extension. json.

Douglas Crockford originally specified the JSON format in the early 2000s.

## 3.LZW Compression

LZW (Lempel-Ziv-Welch) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is designed to be fast to implement, but not necessarily optimal since it does not perform any analysis on the data. It would typically compress large English texts to about half of their original sizes.

The method became widely used in the program compress, which became a standard utility in Unix systems circa 1986. (It has since disappeared from many for both legal and technical reasons.) Several other popular compression utilities also used the method, or closely related ones. It became very widely used after it became part of the GIF image format in 1987. It may also (optionally) be used in TIFF files. LZW compression provided a better compression ratio, in most applications, than any well-known method available up to that time. It became the first widely used universal data compression method on computers.

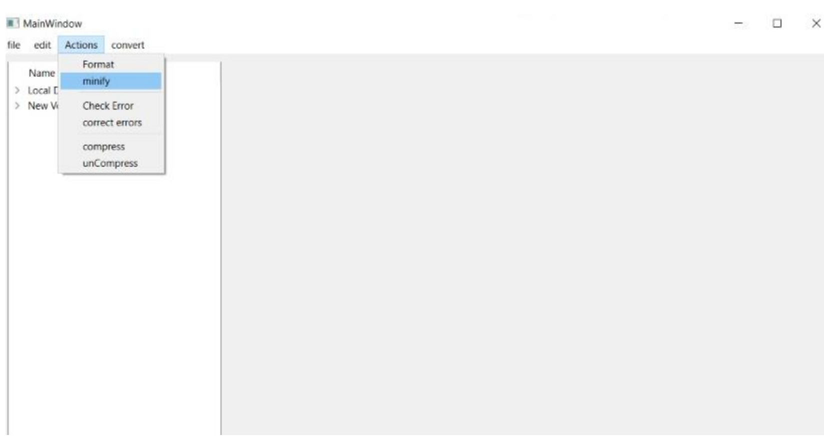
# Implementation Details

Here we are using the print function in the XML tree to format the file

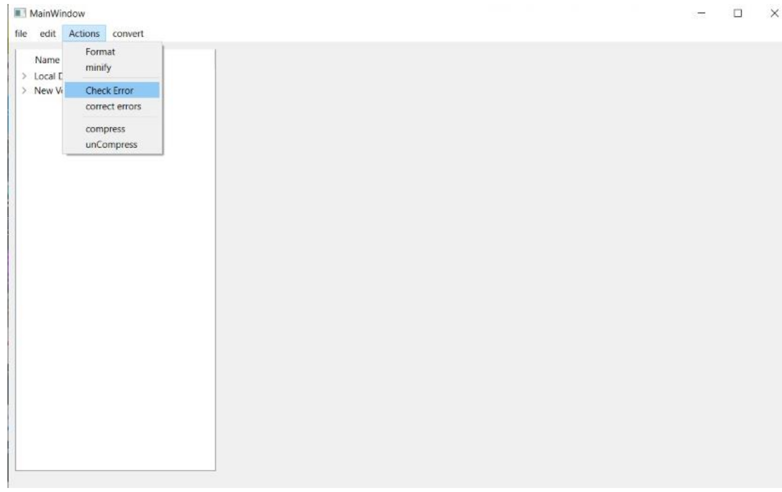
```
void XML_Tree::print(Node * start ,QString &out, int level)
{
         if start equals nullptr
                return
        assign start to Node* current.

        if start->type equal "comment"
            append start data to the output string.
            return

      If there is no attribute

          Add opening tag to the output string.

    Else

          Add the opening tag with its attributes.

   If There are children

        Iterate on each child then call function print.

If current-> data not empty

        Append current data to the output string

Add closing tag

}
```


Here we are using the minify function in the XML tree to delete the tabs, new lines and unneeded spaces the file.

```
void XML_Tree::minify(Node * start ,QString &out)
{
   Minify is the same as print function without tabs and newlines
}
```

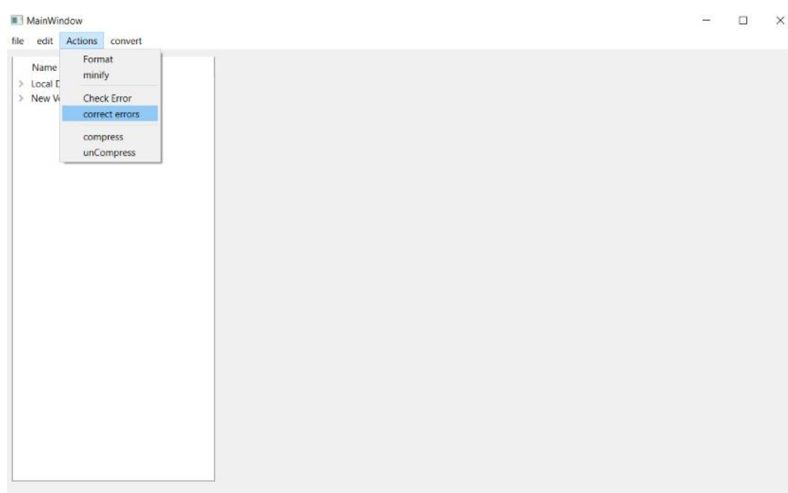


Here we are using the check function in the XML tree to detect the errors in the tree




void XML_Tree::CheckError(Node * node, QString &out,int lvl, QVector<QVector<int>> &high)
{
If node->type equal comment
    Add node data to output string
If no attributes
    Add node -> TagName to output string
Else
    Append attribute
If There are children
    Iterate on each child then call function CheckError
    return
If node-> data is not empty
    append node data to output string
If there is no Node->ClosedTag
    Push index of error to highlight vector
Else if Node-> OpeningTag and Node->ClosedTag don't match.
    Push index of error to highlight vector
    Add closing tag to output string
Else
    Add closing tag to output string
}

Here we are using the check function in the XML tree to correct the errors in the tree

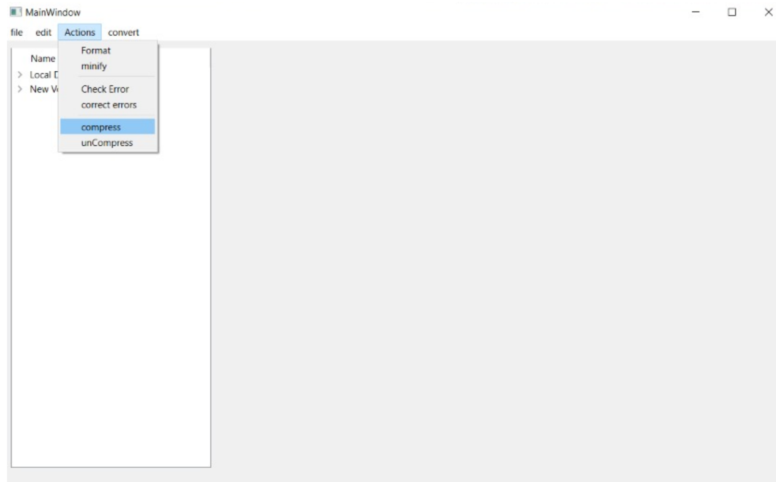


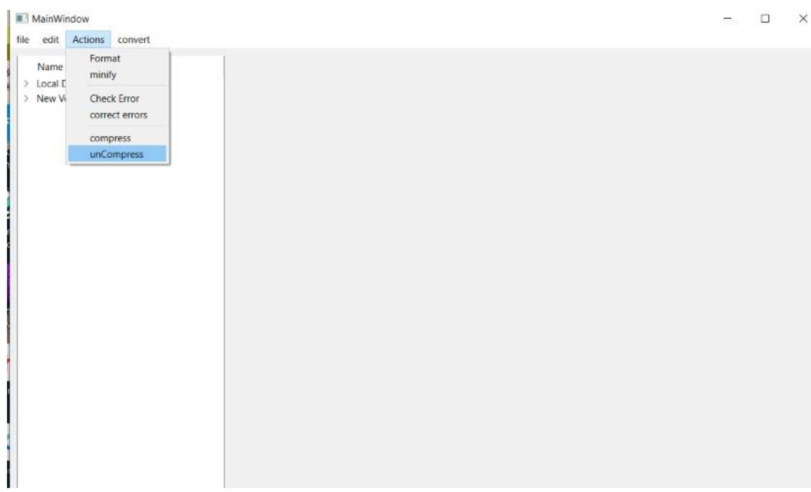

void XML_Tree::CorrectError(Node * node)

{

If There are children
    Iterate on each child then call function CorrectError
If current->type is empty
    return
If current->closed tag is empty or if current->TagName not equals closing tag
    Current->CloseTag equals TagName
}

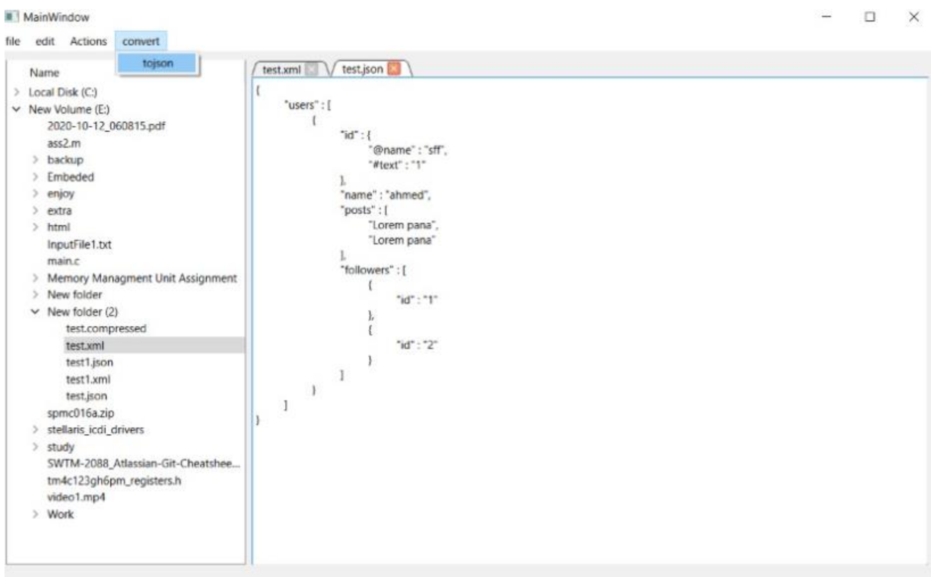
```
template <typename Iterator>
Iterator compress (const std::string &uncompressed,
Iterator result)
 {
Initialize table with single character strings
       w = first input character
       WHILE not end of input stream
            C = next input character
            IF w + C is in the string table
              w = w + C
            ELSE
              output the code for w in result vector
            add w + C to the string table
}
```



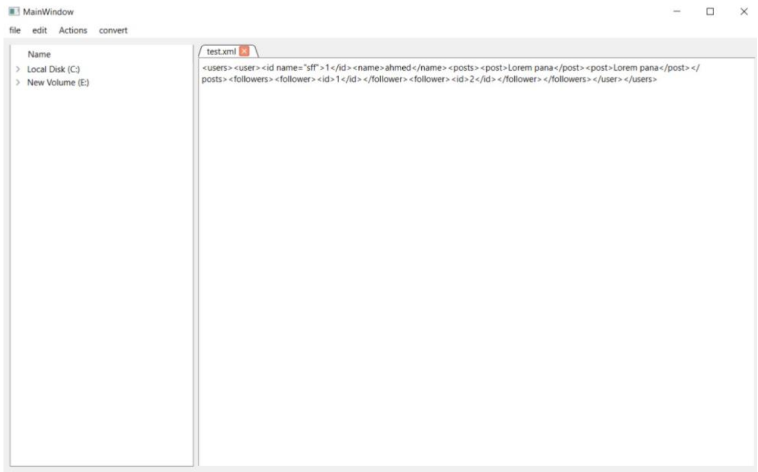




```
template <typename Iterator>

std::string decompress(Iterator begin, Iterator end)

 {

    Initialize table with single character strings
    OLD = first input code
    output translation of OLD
    WHILE not end of input stream
        NEW = next input code
        IF NEW is not in the string table
               S = translation of OLD
               S = S +first character of S
       ELSE
              S = translation of NEW
       output S
       add OLD + first character of S to the string table
  OLD = NEW
  END WHILE
}
```

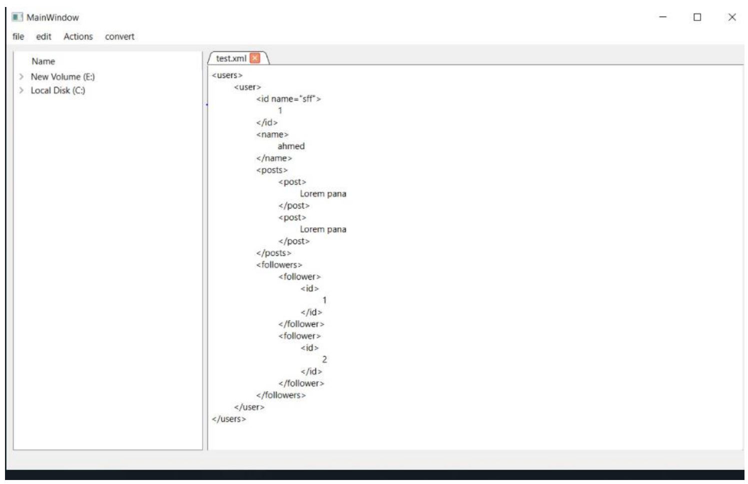
```
void XML_Tree::XMLtoJSON(Node *node, int lvl, QString &outfile)
{
If node is not repeated
   Print node-> tag name
If node has attributes
    Open brackets and print the attributes.
    If node has no children
        Print node->data
   close brackets
Else
    If node has no children
        Print node->data
    Else
      If child->tagname is singular of the node->tagname
         Open bracket "["
      Else
         Open bracket "{"
      Iterate on each child then call function XMLtoJson
      close brackets
}
```



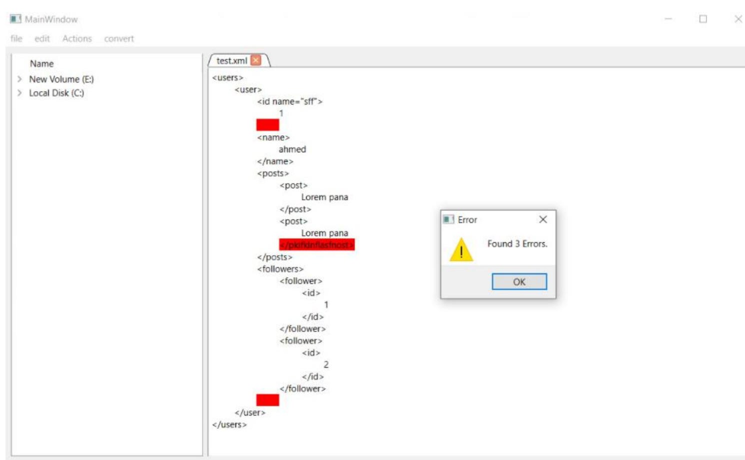
# Usage and Outputs



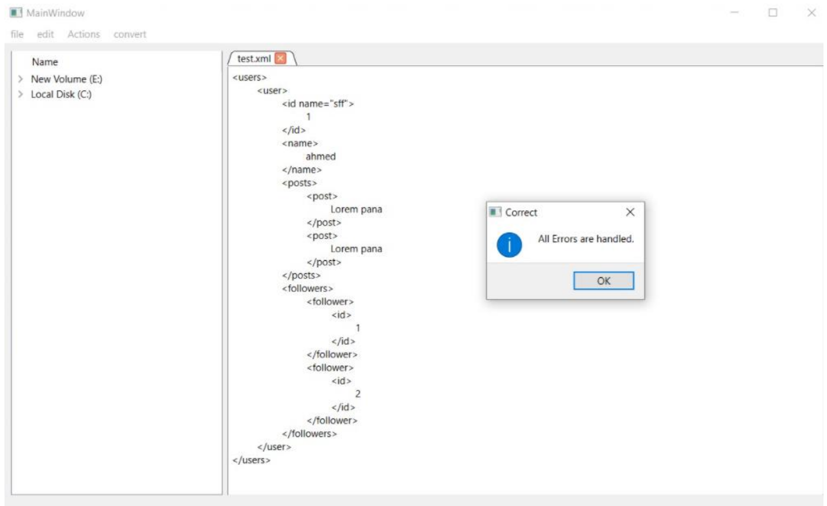

*Minifying*



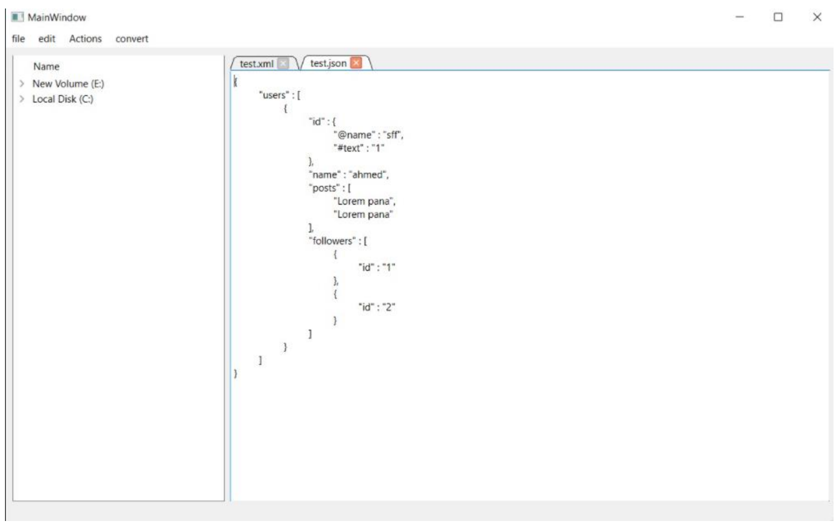

*Formatting*



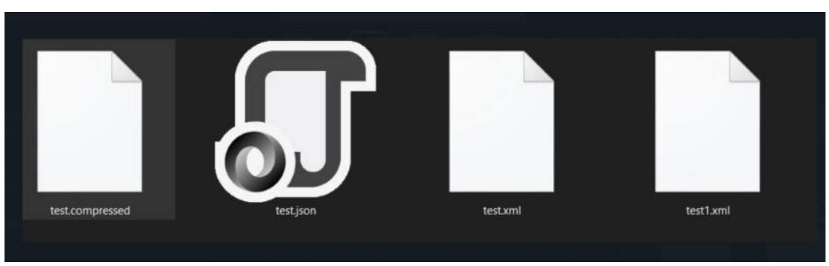

*Visualize Errors*




*Editing Errors*




*Convert XML to JSON*




*Compress and Decompress output Files*

Note :

Test.xml is the original file
Text.json is after converting to JSON
Test.compressed after compressing
Test1.xml is after decompressing

# References


- https://www.xml.com/pub/a/2006/05/31/converting-between-xml-and-json.html?fbclid=IwAR0NlgBQ8vAngpdaVXAGx5C7wQpri4XHQ6_C9jD-kQse5WWFcN3h2Ge_62Y
- https://dev.to/niinpatel/converting-xml-to-json-using-recursion-2k4j?fbclid=IwAR2TbFW_V07sRXwSqBWLUSAAArTAMRgqpIgu5eNPm49mVFN-oQGoHo9VCVc
- https://en.wikipedia.org/wiki/Byte_pair_encoding
- https://www.researchgate.net/publication/289874197_Improving_LZW_image_compression
- https://www.barracuda.com/glossary/data-compression#:~:text=Data%20compression%20is%20the%20process,bits%20than%20the%20original%20representation
- https://www.nhu.edu.tw/~chun/CS-ch15-Data%20Compression.pdf