

Documentation for Application „LogViewer“

Author: Jan Mothes

Lecture: .NET-Programmierung mit C#

Degree program: BA Computer Science

Description:

The LogViewer is a desktop application made to view log file for the purpose of extracting information that is valuable to the user. It should be flexible enough to support log formats unknown to the analyzer through configuration and display the various pieces of information in a structured way.

Outline:

1. Task Description

1.1 Data Sources

1.2 Configuration

1.3 Presentation

2. Implementation

2.1 Structure & Extent

2.2 Third-Party Code

2.3 Architecture

2.3.1 Model

2.3.2 ViewModel State Duplication

2.3.3 File Loader

2.3.4 View

3. User Guide

4. Discussion

4.1 Problems

4.2 Improvements

1. Task Description

1.1 Data Sources

The user should be able to load and display a log file in LogViewer even if an external application is still running and writing to the log file. The user should be able to monitor the file in LogViewer in real-time, meaning that changes to the log file are picked up and reflected in the LogViewer immediately (< 1 sec latency if possible).

When a log file gets cleared by an external application, this should also be reflected in the LogViewer.

1.2 Configuration

Since log files come in various formats and in various structures, the LogViewer should be configurable to be able to read any log file format for structured display.

The user should be able to apply filters before the LogViewer attempts to parse the log data into a structured format, so that unneeded data does not need to be parsed. For example, the user may want to skip any lines that begin with a certain string.

A "log statement" (also referred to as „LogAtom“ in this application) consists of text that was logged to a log file at once, usually caused by a single method call in the external application, and usually has a fixed structure, where each statement first contains several metadata values, followed by an arbitrary log message, optionally followed by additional information like stack traces. Once a statement has ended, it is followed by the next statement on the next line. Usually, the overall structure of metadata is the same in every statement.

To extract the metadata values and bring them into a structured form, the user should be able to specify so-called „Matchers“, where each Matcher is used to extract one kind of metadata value („MetaValue“) from a statement. A MetaValue is defined as a unit of information that semantically belongs together and is present in the metadata of a „Log Statement“.

While the kind of metadata can vary from log to log, some are usually the same in all logs. For example, most logs contain a time-stamp, a log level describing the severity of the information, and a reference to the source which caused the log statement to be printed. For these use-cases especially, easy configuration of required Matchers should be available to the user.

1.3 Presentation

Once the meta-data of all statements has been structured into separate MetaValues, those MetaValues should be displayed to the user in an intuitive way, for example in a table, where each row represents a statement and each column represents a Matcher that was used to extract the MetaValues in that column from the statement.

Furthermore, the user should be able to filter and color log messages by severity (log level), if severity metadata was part of the log file. To be able to do this, the user must be able to specify which of the configured Matchers represents the log level.

2. Implementation

2.1 Structure & Extent

Technology: C#, .NET, WPF

Tooling: Visual Studio 2015, Resharper, Git, NuGet

The solution is split into two projects and applies the architectural pattern MVVM:

- „LogViewer“ which contains Model and ViewModel, as well as some utils functionality
- „LogViewerGui“ which contains only the View (XAML & Code-Behind)

Overview over namespaces in the LogViewer project:

- | | |
|-----------------------|--|
| - LogViewer.Filter | Filters for pre-filtering log lines before parsing |
| - LogViewer.Loader | Loaders for loading data from log files |
| - LogViewer.Matcher | Matchers for parsing statements & extracting MetaValues |
| - LogViewer.Model | Main model (state) & logic of for the application |
| - LogViewer.Utils | Utils, cross-cutting functionality |
| - LogViewer.ViewModel | ViewModel state & logic |
| - .Converter | IValueConverter implementations used in ViewModel |
| - .Matcher | ViewModel containing Matcher settings |
| - WpfUtils | Utils related to common WPF / MVVM functionality |
| - .lib | Third-Party solutions for common WPF /MVVM functionality |

2.2 Third-Party Code

Third-party code is located in the project „LogViewer“ in the namespace „**WpfUtils.lib**“. Code in this folder was not written by the author of this application as can be observed by looking at the references/links to external sources in the source code comments of the code files in question.

The application has several NuGet dependencies:

- **Extended.Wpf.Toolkit** was used for ColorPickers with ComboBox style in XAML
- **MVVMLight** was used for Dependency Injection and as a starting point for MVVM architecture
- **WpfFolderBrowser** was used to provide a much improved folder selection dialog

2.3 Architecture

2.3.1 Model

The top level model class is „LogViewer.Model.LogView“. It contains all the application state which is not specific to UI / view. It contains all raw log lines (for re-parsing when settings are changed), settings (filters & matchers), it detects when settings change and if those changes make it necessary to re-filter/re-parse.

A big part of the model are the matchers. When a bunch of matchers is configured by the user (see „LogViewer.Model.MatcherChain“), each log line is fed through all matchers, which try to parse values out of the log line. When a matcher succeeds, it tells the next matcher in the chain where its value ended, so the next matcher can start parsing the line at that position (see „LogViewer.Matcher.IMatcher“ interface).

One of all configured matchers at a time can be marked as being the matcher that extracts the log level from the metadata (severity). This information can then be used by the ViewModel to style/filter statements based on log level.

Inheritance was used to implement duplicated functionality across multiple matchers. Generics were used for type safety, since a parsed value can be of any type as long as it is comparable and displayable. This makes it possible to implement of type-specific filters on MetaValues like time interval filters for DateTime MetaValues (but this functionality is not yet implemented).

An example for how matchers work:

Given a Matcher chain configured with the following matchers (in order):

1. Date: Column-Name:' Time', Format:'HH:mm:ss.SSS'
2. Text: Column-Name:' Thread', Prefix '[', Suffix ']'
3. Text: Column-Name:' Level', Marked as Log-Level
3. Text: Columns-Name: 'Class'
3. Text: Ommited (Note: used to hide ' - ')

And given the following raw log line:

"17:43:24.186 [thread-1] DEBUG gmm.service.data.DataBase - Users loaded!"

The resulting table would be:

Time	Thread	Level	Class	Message
17:43:24.186	thread-1	DEBUG	gmm.service.data.DataBase	Users loaded!

And the value in the „Level“ Column could then be used to compare against configured Values with associated styles to style rows by log level.

2.3.2 ViewModel State Duplication

When looking at the ViewModel implementation of certain configuration state like Prefilter & Matcher Settings, one may wonder why a lot of state-containing classes are seemingly replicated in the ViewModel, when they also exist in the Model in a similar way. The reason for this is that the data source, if in watching mode, may at any time add additional data on-top of the existing data.

To make this efficient, only new data is re-filtered and re-parsed, while existing log data and the associated extracted structured data stays untouched. But, if the ViewModel state was bound directly to the Model state, any change in configuration would lead to inconsistent state, because the newly added data would be parsed with the new configuration, while the old data would still adhere to the old configuration.

So, to not be forced to re-filter and reparse all data whenever a filter / matcher configuration changes. all configuration changes are temporarily stored in ViewModel state without affecting the Model state, until the user hits the apply button, which then causes all data, old and new, to be re-filtered and re-parsed, so that a consistent state across all data is achieved. The only exception to this are log level settings, which do only marginally affect the Model and do not need any re-parsing/ re-filtering to be applied.

2.3.3 File Loader

The file loader in „LogViewer.Loader“ uses a callback interface very similar to the IObservable and IObservable interfaces that already exist in .NET. Since a little bit more functionality was needed than is available with those, custom interfaces were used, but similarities should make understanding them intuitive.

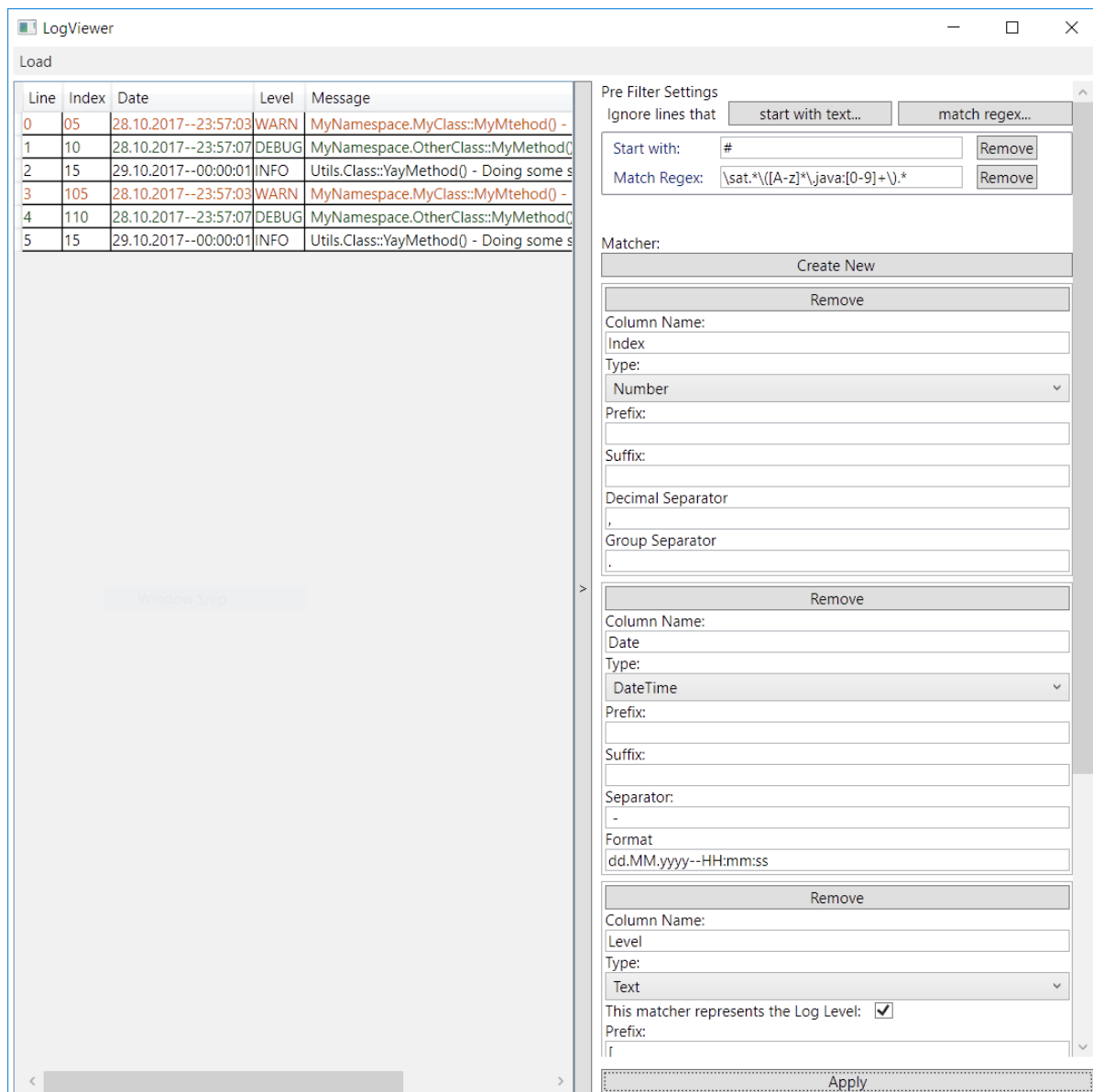
2.3.4 View

The ViewModel is structured such that it reflects the nested structure of the UserControl files of the view for easy DataContext binding. DataTemplates are used a lot since the view contains a lot of dynamic lists / grids.

The main table displaying the parsed values to the users is implemented using a DataGrid bound to a single DataTable object, which is created inside the converter „LogViewer.Model.Converter.LogAtomConverter“. This allows dynamic column headers as needed by the dynamic nature of matchers getting mapped to columns. This dynamic nature also caused RowDetails (a functionality of DataGrid) not being implemented. RowDetails would have allowed to show subsequent message lines and stack trace lines for each log statement, when the user clicks on a row. This information is present in the model for each statement (LogAtom) and could be displayed but it didn't seem like it was possible to provide DataTable with RowDetails information.

3. User Guide

The GUI consists of a menu and two main areas: The log data table on the left side, and settings configuration (which can be hidden) for filtering, parsing and styling on the right side.



Use the Load Button to load a log file into the LogViewer. „Load & Watch“ allows to permanently monitor a file as it is being written to by another application, changes will be represented in the LogViewer automatically.

The configuration area consists of the „Apply“ button at the bottom and 3 distinct parts:

1. „**Pre Filter Settings**“ allow you to filter lines out before parsing them with Matchers.

Pre Filter Settings

Ignore lines that

Start with:

Match Regex:

2. „Matcher“:

Here you create a matcher per table column. Every matcher will for every statement extract a value from that statement as configured. By default, most matchers will extract a word until the next whitespace occurs. By specifying a Separator value, those matchers can be configured to stop extracting at any string pattern, not just whitespace.

Some matchers can be configured to have a Prefix/Suffix. This can be used to remove unwanted parts from the beginning/end of a matched value.

The „DateTime“ Matcher must be configured using a DateTime format string, see <https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-date-and-time-format-strings> for exact syntax.

The Regex Matcher must be configured with a regular expression, and this expression should contain a capture group to specify the value to be extracted from the matched part of the string.

The Number matcher will simply match a number until it finds no more digits, and the Text matcher will match any text until the given Separator is found (or until whitespace, if Separator is not configured).

Each matcher has a name, which corresponds to the column name in the table area and should describe the meaning of the associated values.

Matchers which are used to extract values of type string (Text & Regex) can be marked as representing the log level. Those matchers typically extract values like „DEBUG“ or „INFO“, and this can be used in the third configuration area to style / filter statements by log level.





Matcher:

Create New
Remove
Column Name:
Level
Type:
Text
This matcher represents the Log Level: <input type="checkbox"/>
Prefix:
[
Suffix:
]
Separator:
-

3. „Log Level Settings“

If a configured matcher was marked as Log Level, the values it extracts will be compared with the Names of the configured Levels in this area. If they are equal, the statement which contains the Level will be filtered as configured in the „Show“ column and colored as configured in the „Color“ column.

Log Level Settings

Add Level			
Show	Name	Color	
<input checked="" type="checkbox"/>	DEBUG	 ▼	Remove
<input checked="" type="checkbox"/>	INFO	 ▼	Remove
<input checked="" type="checkbox"/>	WARN	 ▼	Remove
<input checked="" type="checkbox"/>	ERROR	 ▼	Remove

4. Discussion

4.1 Problems

Sadly due to the dynamic nature of the binding to DataGrid, implementing the display of additional message lines and stack trace lines for each statement row proved to difficult to implement (see 2.3.4). The author could not find any resources describing how to achieve the wanted result nor was it intuitively clear how to implement this feature.

4.1 Improvements

While the basic functionality was successfully implemented, there would be multiple ways to leverage the the ability to parse log files into structured data structures. For example, it could make sense to export the information to a SQL database, since only minor mappings issues (selecting the right data types) would have to be considered.

Also, more complex filtering depending on the type of extracted values and search functionality would make it easier to analyze big log data.

The model implementation could probably be optimized to handle matcher / filter changes more efficiently (only update statements that have to be updated), but this would come at the cost of much increased complexity in the filtering / parsing logic, because a change in the parsing of a log line can affect the parsing of the previous and next log line (parsing success/failure can determine whether a log line is interpreted as being the start of a new statement or part of the statement that came before). Thus, any per-line or per-statement updating is non-trivial, if parsing is affected.