*Log Inspector in Python*

Table of Contents

[Introduction 2](#_Toc72338293)

[Scope 2](#_Toc72338294)

[Process Flowchart 2](#_Toc72338295)

[Pre-requisites 3](#_Toc72338296)

[Setup 3](#_Toc72338297)

[Conclusion 17](#_Toc72338298)

Introduction  
  
Within software development, application logging plays a key role. As much as we’d like our software to be perfect, issues will always arise therefore it’s important to have a robust monitoring and logging in place to control and manage the inevitable chaos.  
Nowadays, application support engineers need to be able to easily access and analyze the huge volumes of log data their applications and infrastructure are generating. When an issue occurs, they cannot afford to wait for a minute or two until a query returns results. They need speed, regardless of the amount of data they are collecting and querying.

Scope  
   
In this tutorial, we will explain how to detect changes in a particular directory (Let’s suppose the directory hosting the logs of your application(s)). Whenever a change occurs, the modified or newly created files of predefined types will be processed in a timely manner to retrieve the lines meeting specified patterns. On the other hand, all the lines within these files that don’t match the specified patterns are considered outliers and they are abandoned in our analysis.

To speed up the analysis of the vast amount of data that might be contained within the files to be monitored, we will develop a visualization tool allowing you to pinpoint, bang on time, any anomaly or observation.

We will leverage several components like Redis, Flask, SocketIO and watchdog to detect the changes occurring and to send a push notification to the client signaling any observation that might be identified.

# Process Flowchart



Triggered Events

New  
File

On\_Created

Deleted File

Updated File





On\_Modified

Log Handler

Log Watcher

Folder to monitor

On\_Deleted

# Pre-requisites

As our requirements stand, the following components come into play:

1. Redis is an open source, advanced key-value store and an apt solution for building high-performance, scalable web applications.  
     
   Redis has three main peculiarities that sets it apart.

* Redis holds its database entirely in the memory, using the disk only for persistence.
* Redis has a relatively rich set of data types when compared to many key-value data stores.
* Redis can replicate data to any number of slaves.

Installing Redis is outside the scope of this tutorial however I would suggest to follow the steps mentioned in the given link for installing Redis on a Windows environment:  
[redis - Installing and running Redis Server on Windows | redis Tutorial (riptutorial.com)](https://riptutorial.com/redis/example/29962/installing-and-running-redis-server-on-windows)

NB: For the purpose of this tutorial, I am using Redis version 3.0.504.

1. Socket.IO isa JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a server-side library.
2. Watchdog is a cross platform Python API library and shell utilities to monitor file system events. As the name suggests this module observes the given directory and can notify if any file change occurs.
3. Pygtail is a Python library enabling to read file lines that have not been read. It will even handle files that have been rotated.
4. Flask is a web application framework written in Python.

This tutorial will be presented within a Flask framework; however, it could easily be ported to other Python frameworks (Django, Pyramid).   
If this tutorial intrigues you and makes you want to dive into code immediately, then go to this Github repository “<https://github.com/bassemmarji/Flask_Log_Inspector>” for the code used in this article.

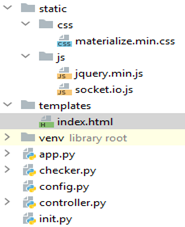
# Setup

Setting up is quite simple and straightforward. Of course you need python3 installed on your system. It is highly recommended to setup a virtual environment where we will install the needed libraries.

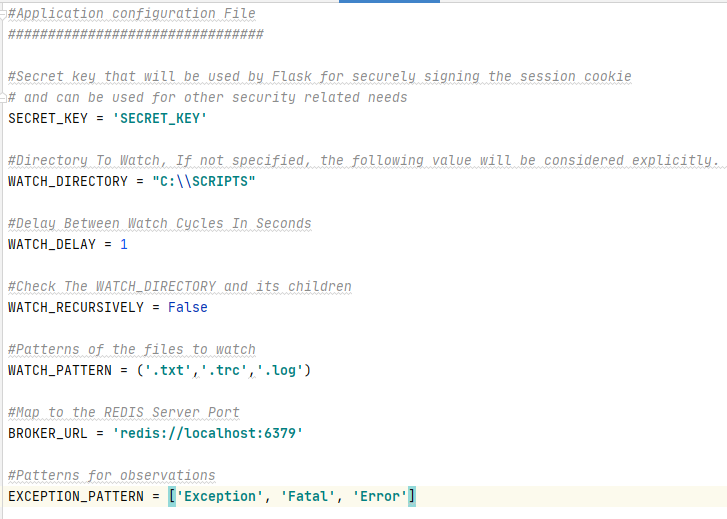
1. Create a virtual environment and activate it.
2. Create a file named requirements.txt and add the following lines in it.

|  |  |
| --- | --- |
| requirements.txt | The following figure will show you the installed libraries and their dependencies |
| Flask==1.1.2 Flask-SocketIO==5.0.1  Pygtail==0.11.1  redis==3.5.3  watchdog==2.1.1 |  |

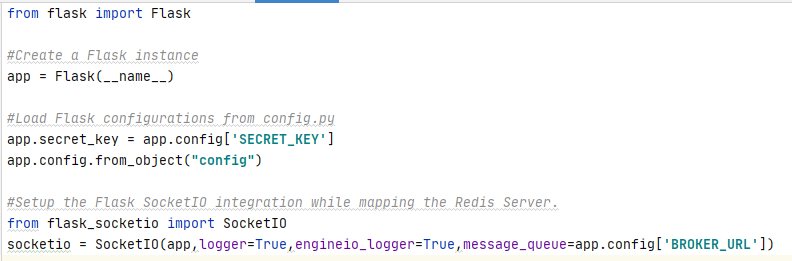
1. Now, let’s install these libraries to the project.  
   pip install –r requirements.txt
2. Create a folder for hosting our project called “Flask\_Log\_Inspector”.

At the end, our folder structure will look like the following:  
  
  
  
Let’s start writing the actual code.

1. Let’s define the configuration parameters for our application within config.py:

*# config.py* ****

**NB: For the sake of brevity, I hardcoded these configuration parameters in “config.py” but it’s recommended to store these parameters in a separate file (for example .env).**

1. Create an initialization file for our project:  
   *#init.py*  
   
2. Define a checking mechanism:

This mechanism will leverage the modules “PygTail” and “RegEx” in order to pinpoint the observations based on the parameter “EXCEPTION\_PATTERN” defined in “config.py”.

The function “**checkForException**” defined in this program will ingest the events dispatched by the observer class of the watchdog module (to be seen later on).  
  
Basically these events will be triggered for any file change in the given directory.   
  
An event object has 3 attributes:

* **event\_type**: The type of the event as a string (modified/created/moved/deleted).
* **is\_directory**: return True if the event was emitted for a directory.   
   return False if the event was emitted for a file.
* **src\_path**: The source path of the file system object that triggered this event.

*#checker.py*



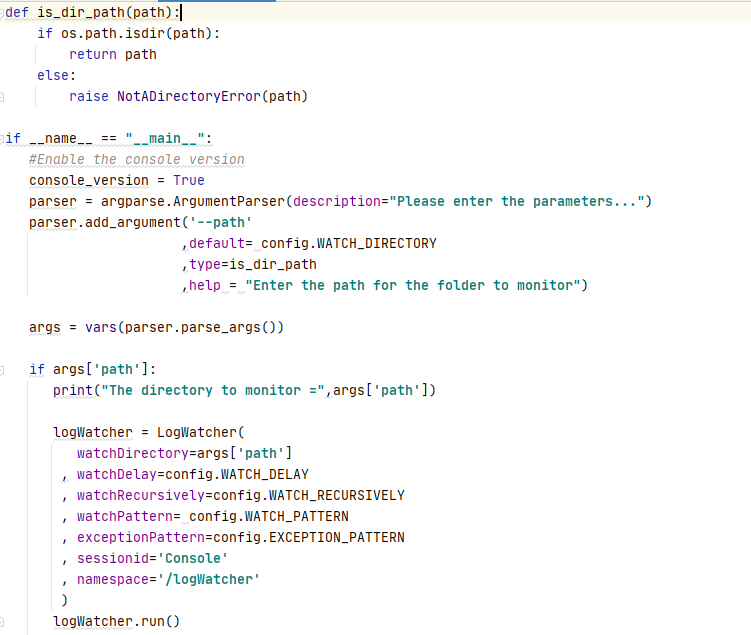
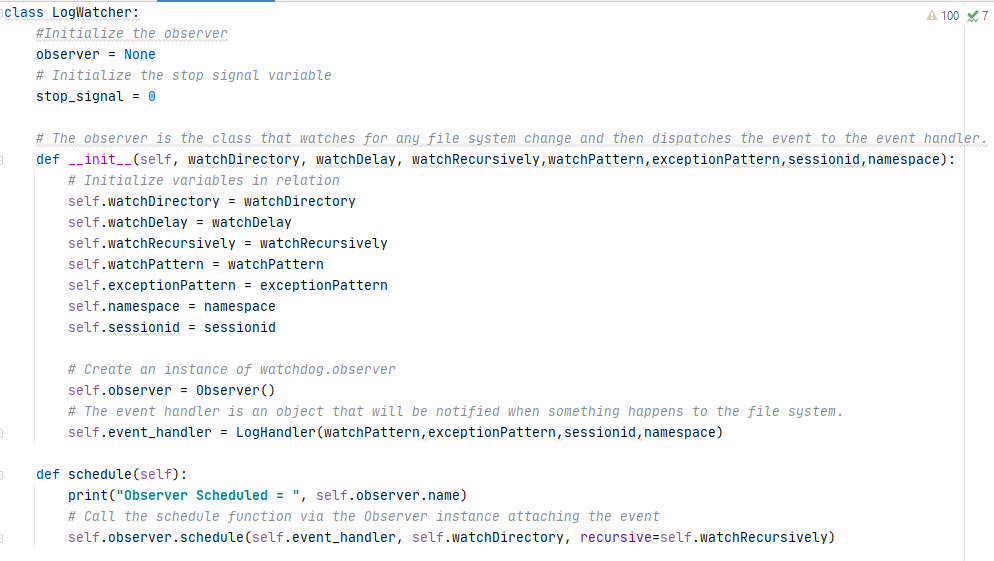
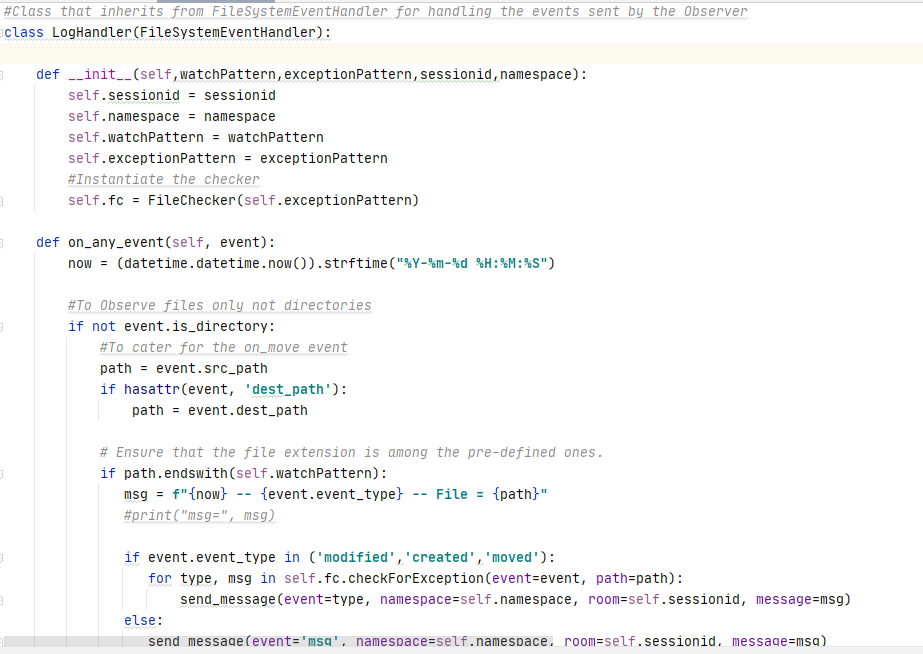
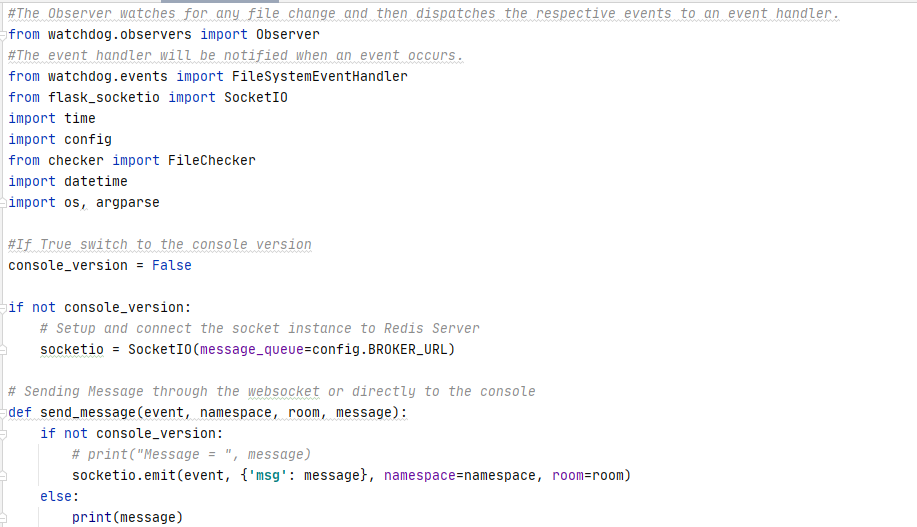
1. Define a controller:  
     
   The main building blocks of our controller program are the following classes:  
   * **LogWatcher:** 
     + Create an instance of the watchdog.observer thread class.  
       The observer watches for any file system change and then dispatches the corresponding event to the event handler.
     + Create an instance of the Event Handler “LogHandler” that inherits from the FileSystemEventHandler.  
       The event handler will be notified when changes occur.
     + Assign a schedule to our observer and define other input parameters like the directory to watch and the watching mode (Recursive or Non-Recursive).  
       Beware when setting the recursive parameter to “True”, ensure that you have sufficient access rights on the subfolders.
     + Run the observer recurrently taking into account the lapse (“Watch\_Delay” specified in the “config.py”) at every iteration until the variable stop\_signal is set to 1.  
         
       Once started, the observer will run in a reserved thread and it will emit messages across the web-socket to signal its startup.

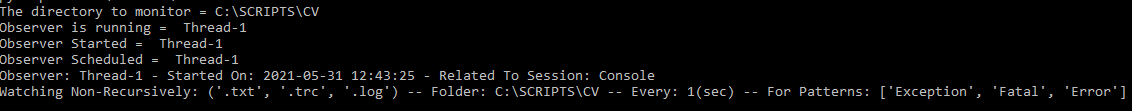
The same applies when the observer is stopped and hence its reserved resources will be cleaned up.

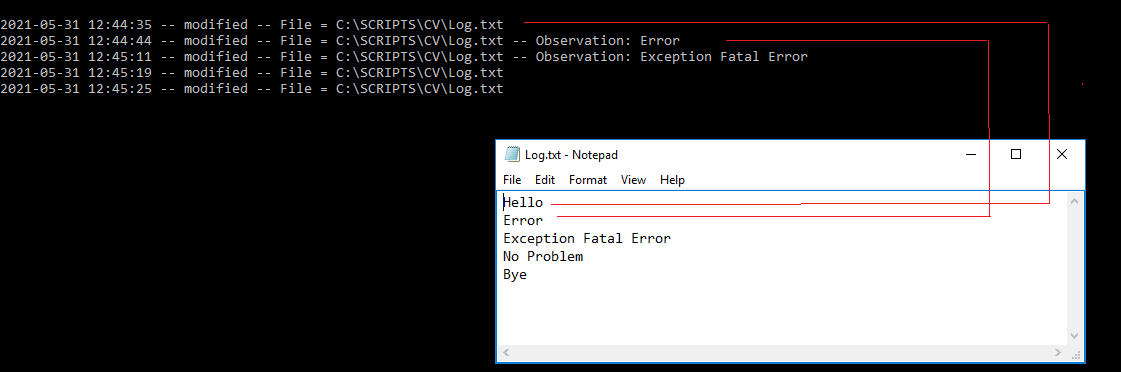
* **LogHandler:**  
  This class inherits from the class named “FileSystemEventHandler” of the watchdog library and overwrites mainly its function “on\_any\_event”.  
    
  Below are some useful methods of this class:  
  + on\_any\_event: Called for any event.
  + on\_created: Called when a file or a directory is created.
  + on\_modified: Called when a file is modified or a directory renamed.
  + on\_deleted: Called when a file or directory is deleted.
  + on\_moved: Called when a file or directory is moved.

The code allotted for the “on\_any\_event” function will:

* + Observe files only not directories.
  + Verify that the extension of the file subject to the event is among the ones pre-defined in the variable “WATCH\_PATTERN” within the program “config.py”.
  + Produce a message illustrating the event or the observation if detected.

*#controller.py  
*Now in order to make sure that you are moving on the right track and before digging further let’s test the console version of our program “controller.py” in a lightweight mode:  
  
To do so, open up a terminal window and type the following in it:  
python controller.py --path “C:\Scripts\CV”

You should see an output similar to the following:  
Go to the folder subject to monitoring and perform the following:

* Create a new file i.e. Log.txt.
* Add a new line (i.e.: Hello) and save the file. In this case no observation will be detected, however a message will be displayed.
* Add a new line (i.e.: Error) and save the file; An observation will be highlighted as shown below:   
    
  
* Delete the file Log.txt and press CTRL-C to exit:  
    
  

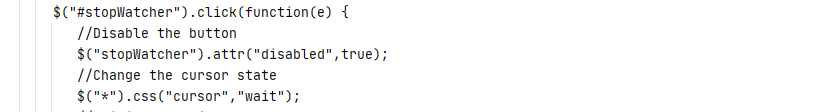
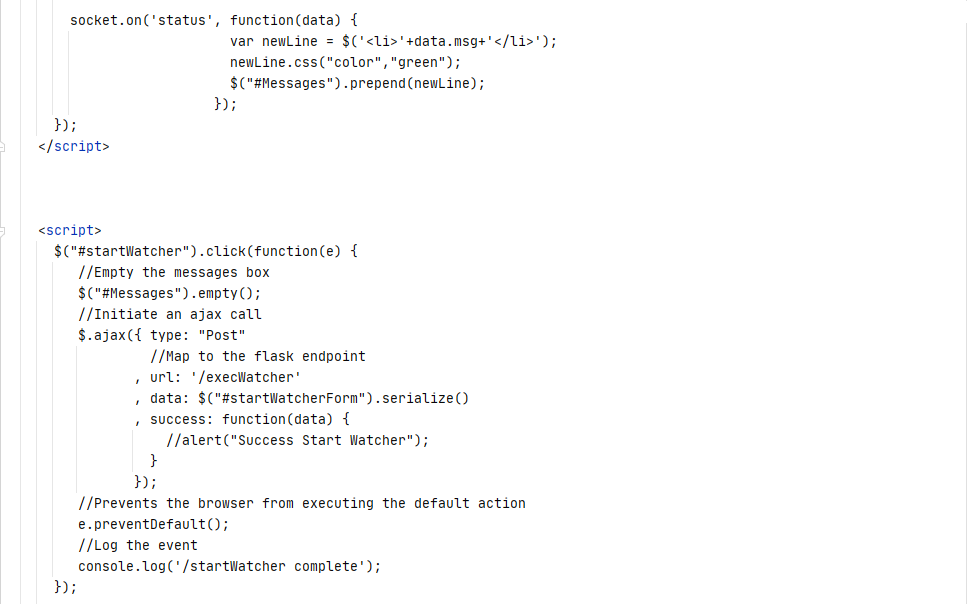
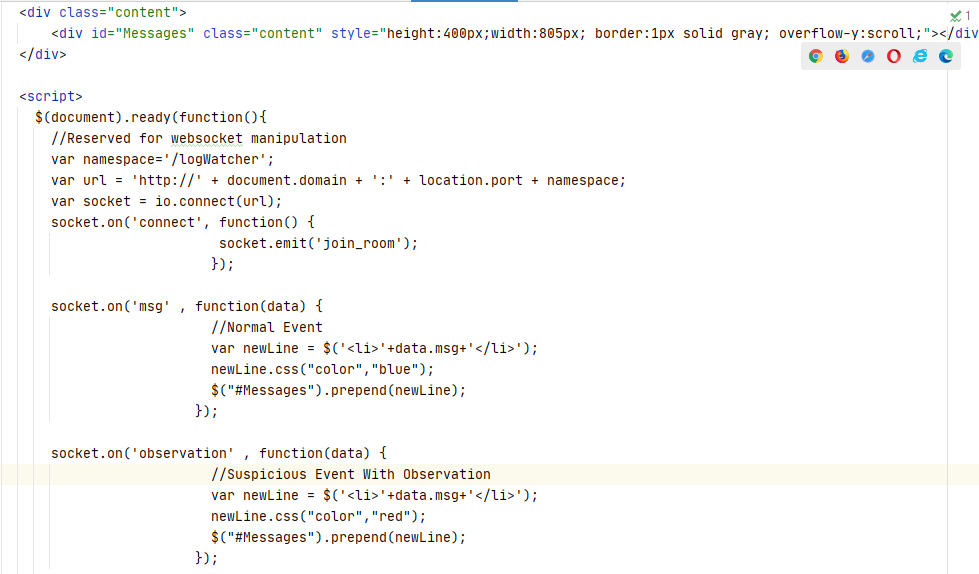
*Worth noting the following:*

* The [argparse](https://docs.python.org/3/library/argparse.html#module-argparse) module used in this program makes it easy to write user-friendly command-line interfaces.   
  Using this module we added an optional argument “—path” to our command line interface and we defaulted the value of this optional argument to the parameter “WATCH\_DIRECTORY” specified in the configuration file “config.py”.
* We defined the function “is\_dir\_path” and associated it with the argument “—path” in order to ensure that the path entered is a valid directory.
* The KeyboardInterrupt exception added within the function “run” of the class “LogWatcher” is raised when you try to stop a running program by pressing ctrl+c or ctrl+z in a command line.

1. Build the Flask framework and define the routes:  
   *#app.py*  
   

In brief, we performed the following steps:

* We defined three routes within this program:  
  + “/” 🡺 create a unique session ID and store it within the Flask session then render the web page (index.html).  
    The unique session ID will serve to allocate an exclusive room for each user when dealing with the web-socket communication.
  + “/execWatcher” 🡺 launch the observer and wait for events.
  + “/getAsyncTaskResult” 🡺 stop the observer.
* To manage effectively our connection over the web-socket:  
  + We will set the value “/logWatcher” for the namespace. (Namespaces are used to separate server logic over a single shared connection).
  + We will assign a dedicated room for each user session. (Rooms are subdivisions or sub-channels of namespaces).

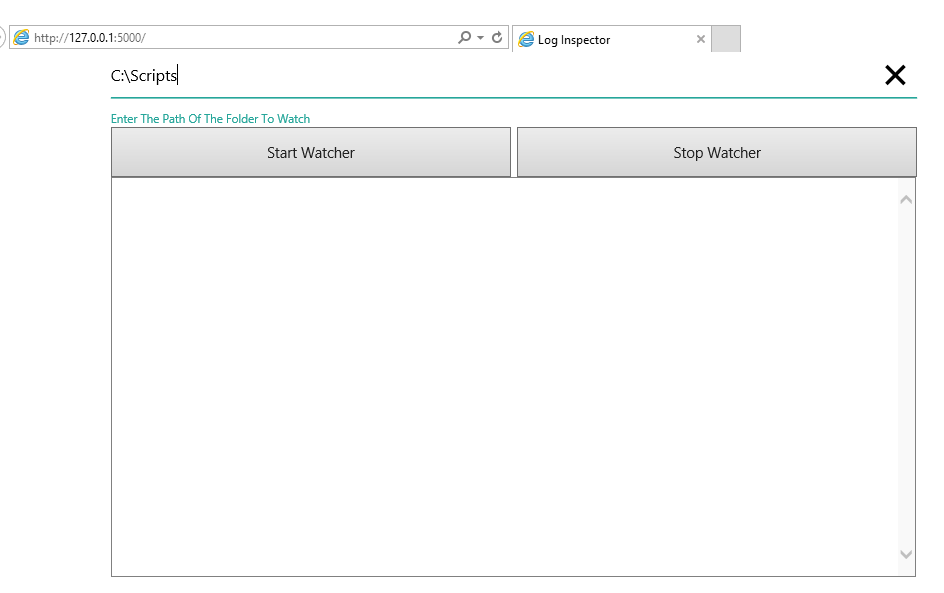
1. Next, let's build the front-end for our web app. Create a file named index.html in the templates directory.  
   *#index.html*  
   **

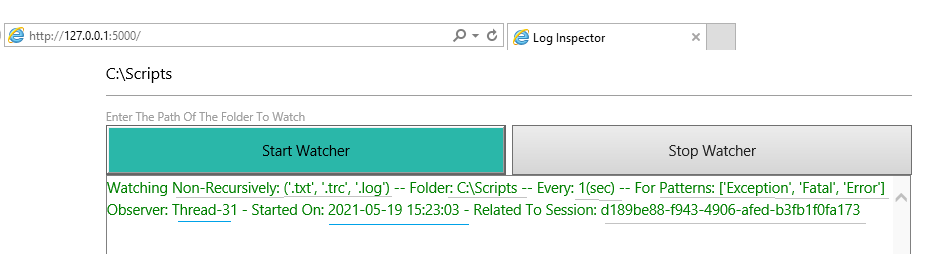
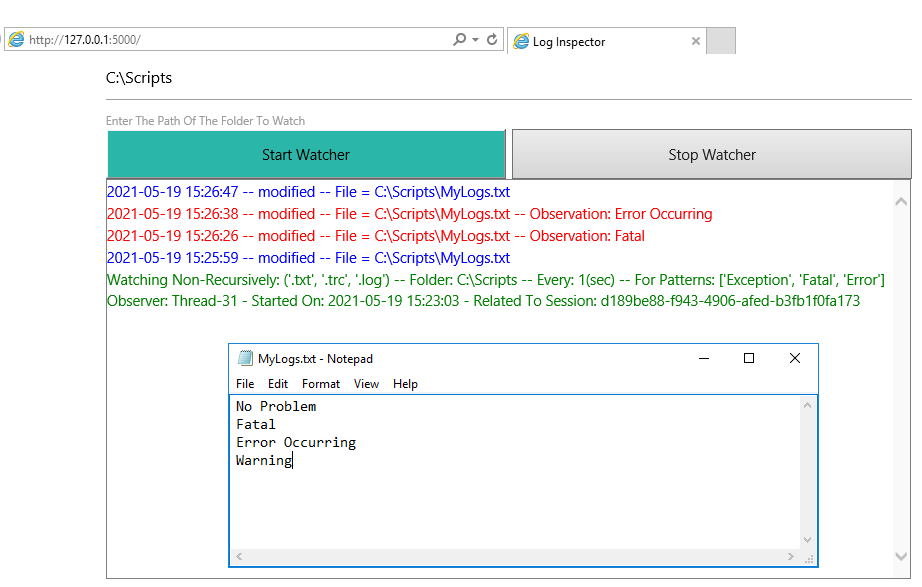
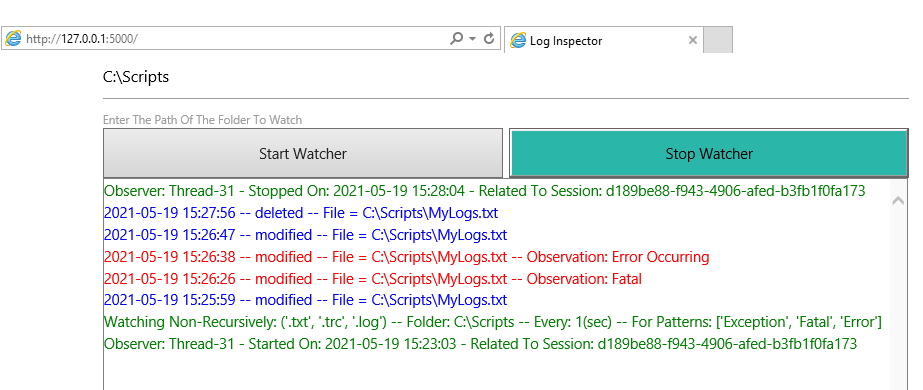
This template will include the following:

* A field “inpFolder” representing the path of the directory to monitor.  
  Worth noting that in case we left this field empty, the path defined in the parameter “WATCH\_DIRECTORY” within “config.py” will be considered.
* The button “start Watcher” for launching the observer.
* The button “stop Watcher” for stopping the observer.
* The section messages for displaying the events status.

Let’s test our program:  
  
Please proceed as per the following steps:

1. Start “Redis” server  
   On Windows locate the folder where “Redis” is installed and double-click on “redis-server.exe”.  
   For a default installation, please make sure that a “Redis” instance is running on TCP port 6379.
2. Open up a terminal window and type the following in it:

**python app.py**Then open your browser, access the following link and enter the path of the folder to observe:  


* By pressing on the button “Start Watcher” an observer will be activated on the given folder.  
  A message will be displayed in the “Messages” section showing the ID of the observer and other related info.  
  
* Go to the folder subject to observation and perform the following:
* Create a new file i.e. MyLogs.txt
* Add a new line (i.e.: No Problem) and save the file. In this case no observation will be detected, however a message will be displayed.
* Add a new line (i.e.: Fatal) and save the file.   
  An observation will be highlighted and the respective line will be displayed in red color in the Messages section.  
  
* Go to folder under observation and do the following:
* Delete the newly created file “MyLogs.txt”.
* Press on the button “Stop Watcher”.  
  

Conclusion

I hope this article has helped you after we explored in depth the available features of the “watchdog” and the “pygtail” libraries.  
It is worthy to note that by extending the described functionalities, you may associate an alerting mechanism or incident workflows with the configured exception patterns. By doing so, when an observation is pinpointed the configured workflow or alert will be triggered automatically.

|  |  |
| --- | --- |
|  | ***Bassem Marji*** *is a project implementation manager at BLOM Bank with a proven track record of success.  He managed the implementation of over 50 projects and propelled the digital transformation of mission critical applications. He spends his free time discovering the latest technology trends in the IT field.* |