First of all. The code does not build…

In method LogMessage we have two parameters with the same name ("message") but different types.

27: public static void LogMessage(string message, bool message, bool warning, bool error)

So, let's assume that string message is the text we actually want to log, and, for what I can see, bool message tells this component whether to log that text as an information message or not.

Another thing that is breaking the build is that there are two variables that are used without been properly assigned.

47: int t;

64: string l;

Since both variables are assigned depending on values pass to the constructor, it is completely possible to find a combination of values that leave both variables unassigned, for instance:

logError = true;

logMessage = false;

logWarning = false;

message = true;

warning = true;

error = false;

That combination of values will leave t unassigned.

Besides that, another minor detail I found is that there is the variable \_initialized that is never used.

Once this issues are fixed we can study the state of the code and the way this component was designed.

The first big thing I noticed is that the method LogMessage is declared as static and it uses some static properties from class JobLogger. The funny part is that these static variables are assigned with the instance constructor! This means that in order to use the logger one needs to create a useless instance of JobLogger so as the static variables are initialized. Since static members are global to the running application and live until the application finish, if I create a new (again useless) instance of JobLogger with a different combination of values, the original set of values will be overwritten for every call to LogMessage that we have in the application.

My first suggestion with this item is to transform all the static members of JobLogger into instance members, so we may have multiple and independently configured instances of the logger.

Okay, let’s look at the responsibilities of our logger. It seems to know how to writes messages to the console, writes messages to the database and writes messages to a file. But, there are a lot of things that this little class does if we also consider that it knows how to open a connection to the database, how to issue the query that inserts the message in the logs table, how to operate the file system in order to read/write the message in a logfile… I' seems that this class does too many things within a single method. Now, let's suppose that in the future we need to add new logging methods, let say we need to send the message to a RESTfull end-point, send the message of type error via Email and/or add an entry into the windows's event log… If we add all this logic into the same LogMessage we'll end up having a kind of god object that knows how to do everything (and let's not talk about how gigantic the class will be).

My suggestion in this case will be to separate the responsibilities in different classes… one for log into the console, one for log into the DB, and so on. We can consider implementing a Composite pattern that help us to glue up everything together and still gives us the flexibility and extensibility we need. A builder patter could also be taken into account for easing the construction of the composed object.

As for what type of message should be logged, we can move that logic to an abstract class from which each particular logger can inherit.

Now, I can see that some of the settings (like the path to the logfile or the connection string) are taken directly from the ConfigurationManager.AppSettings. If we want to have a truly versatile logger, we can't tide it up to get the settings from a single and specific source. There are many ways to store settings, I've seen applications retrieving them from a database or requesting them to exposed services. I would prefer not to couple the logger with a particular setting provider. The best choice will be to make the logger take this required values from the constructor. This way the logger doesn’t have to care from where it gets its configurations.

If we are going to use ConfigurationManager to retrieve the connection string, I would suggest to take it from

ConfigurationManager.ConnectionStrings instead of AppSettings. This way we use the section of the App.config (or web.config if it is web application) intended to contain this kind of settings.

Let's check the logic of each type of logger:

Log to the Console:

The behavior seems fine, although before writing the message I would switch back to the default foreground color so the logger does not interfere with another component that might be writing stuff to the console.

Log to a File:

First of all, the code checks if the file exists, and if it DOES NOT It tries to open it anyway and read it. This will never work if the file doesn't exist. The correct behavior would be to check if the file exists and if it does, open it, otherwise, create an empty file.

Also, it is not good to read the whole file to memory just to add a few words and then overwrite the whole file. Imagine having to log something into a file with a size of 1GB, the logger will surely eat up all the memory (if not throw an OutOfMemoryException). It is simpler and most efficient to open a FileStream and write to it.

Both of the previous items could be easily solved by using File.AppendText(...)

Another thing to notice is the name of the file. It contains the file creation date in a format that could vary according to the culture configured in the running thread. This could issue an inappropriate or invalid filename. I suggest to change it to a more neutral format like 'yyyyMMdd'

Log to Database:

We can remove variable t and use directly the enum LogMessageType messageType that is passed to the method LogMessage and avoid writing a bunch of ifs. I'm assuming that the values of the enum can relate to the ones intended to be store in that table. What I mean is that for instance LogMessageType.Warning has a value of 2, and we know that a 2 stored in table log means a warning message. But if instead table Log had a referential constraint to another table, let's call it MessageTypes, in which a Warning is represented by value 3, then this approach would not work; and we should consider finding some kind of mapping from one type of value to the other. For now, given the context of this test, I will take the simpler option and assume that both values have the same meaning.

I notice that the insert statement doesn't include a field list, nor a timestamp either. I could assume that the timestamp is a calculated column in that table, but since I don’t have the table's definition I will assume that that field doesn't exist. I suggest to include that column in the table and also specified the column list in the insert statement.

Another thing to notice is that the connections is never closed. We should wrap this whole code in a using statement so it will call connection.Dispose() automatically and release every resource it could have taken. Also, the SQLCommand is never associated with the connection.

Another good practice is to use query parameters instead of hardcoding the query string. This way we can avoid having to cast and format each field, and we are protected against injections.