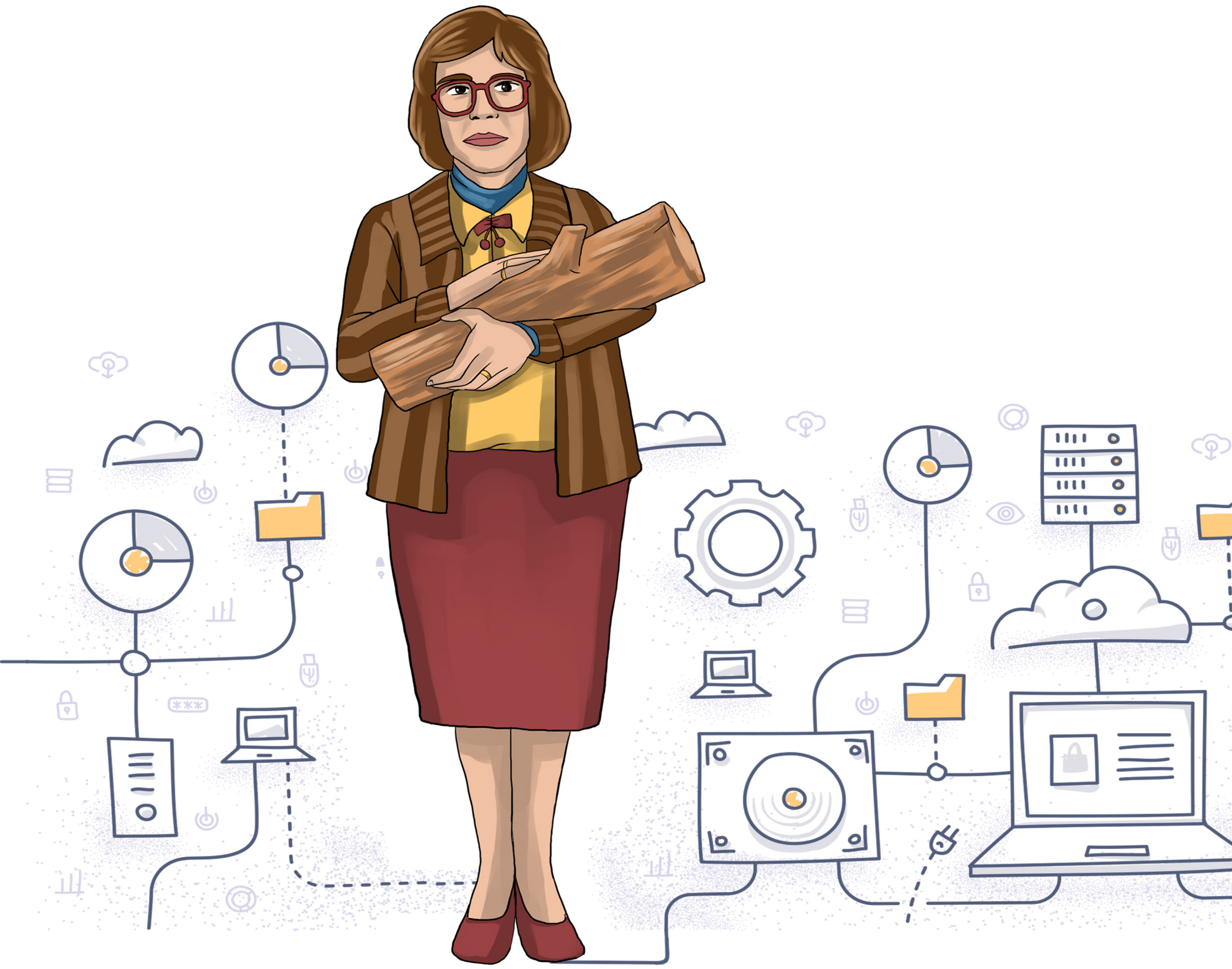
.NET WEB APPLICATION LOGGING ESSENTIALS

A practical guide to logging in ASP.NET, MVC, Web API and Core



THOMAS ARDAL

.NET Web Application Logging Essentials

A practical guide to logging in ASP.NET, MVC, Web API and Core

Thomas Ardal

This book is for sale at http://leanpub.com/net-web-application-logging-essentials

This version was published on 2018-05-22

Contents

Introduction

Welcome to this ebook about logging. During the last 15 years, I have been building desktop and web applications in a range of different programming languages and application frameworks. Since Microsoft released ASP.NET MVC back in 2007, I have been working with Microsoft technologies exclusively. This book is a summary of what I have learned about logging during my career as a professional software developer.

Logging may be one of the most overloaded terms in modern software development. When thinking about logs and logging, you may be thinking about long text files spread across servers. After reading this book, I'll show you a better solution. With the introduction of cloud solutions like elmah.io and semantics logging frameworks like Serilog, things have changed a lot during the last 5 years.

Throughout the rest of this book, I'll use a range of terms. Here's a quick overview of the most common terms and what they mean:

Log message: A log message is a standalone piece of information like a message logged through a logging framework, an error logged through a framework like ELMAH, etc. A log message may or may not include additional information like a timestamp, a severity, custom properties, etc.

Log: A log is basically a container for log messages. A log typically bundles log messages from the same application or the same environment. A log can be kept in a file, inside a cloud service, etc.

Logging framework: A logging framework is a library able to write log messages to one or more logs. We'll go through a range of popular logging frameworks in the following chapter.

Application Logging

We've compiled a list of logging frameworks for .NET. There's a great deal of options out there, but a lot of them are no longer maintained. The list provided in this section contains the popular choices. Different frameworks support different features, however all frameworks contain a similar API for storing new log messages. Before we start digging into each framework, let's discuss some common features and terms.

Logging API

In order to store log messages, the logging framework provides you with a set of APIs. Most logging frameworks contain the concept of severities. When logging a message, you will need to tell the framework how critical the log message is. While named differently, it's common to provide different severities, from debug to critical messages.

All logging frameworks support some kind of configuration API to tell the framework where to keep log messages. This is often referred to as appenders or targets. The older frameworks typically configure this through XML, while the newer frameworks support configuration through a C# based fluent API.

Logging Additional Information

In order to help with debugging, you may want to attach additional information to a log message. Rather than just extending the actual string logged as part of the message, most logging frameworks support a way to attach properties to a log message. Beneath each logging framework, I'll show you how to attach additional information to a log message.

Semantic/Structured Logging

Newer logging frameworks support the concept of semantic or structured logging. Using semantic log messages, you embed properties directly in the text message. Embedding information in the message like this has great benefits. With logging frameworks that do not support semantic logging, you may combine strings like this:

```
log.Information("{0} says hi", "Dolores");
```

This will output `Dolores says hi` in your log file. Your logging framework doesn't know what `{0}` represents, other than to switch the name `Dolores` in there.

Using semantic logging, your log message would look like this:

```
log.Information("{Firstname} says hi", "Dolores");
```

Here, we switch to using a named variable (`Firstname`), but your output in a text file would look the same. The advantage of putting semantic meaning inside your log message reveals itself when using something like a NoSQL database, or elmah.io, as your target. Here, you will now be able to search log messages using variables:

```
FROM log WHERE Firstname == "Dolores"
```

Another benefit is that your logging framework now knows that `Dolores says hi` and `William says hi` actually represents the same log message. This can be used to group identical errors together and to detect new errors.

Logging Frameworks

log4net

Probably the oldest logging framework on the block, log4net has existed pretty much since .NET was introduced. log4net was originally an internal Apache log4j port developed by Neoworks Limited back in 2001. The project quickly moved to Sourceforge (the GitHub of 2001) and was released under the Apache license. Since that time, log4net has been one of the most popular choices in the .NET world for adding logging to applications.

log4net works with the concept of appenders, where log messages can be routed to different data stores. A lot of appenders have been implemented during the years, like logging to the file system, SQL Server, HTTP endpoints, and even NoSQL databases, however being unstructured text messages, log4net and NoSQL don't exactly go hand in hand.

log4net is configured through your `app.config` file or a dedicated `log4net.config`:

```
<log4net>
  <appender name="FileAppender"
            type="log4net.Appender.FileAppender">
    <file value="log-file.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern
        value="%date %level %message%newline" />
    </layout>
  </appender>

  <root>
    <level value="ALL" />
    <appender-ref ref="FileAppender" />
  </root>
</log4net>
```

Log messages are written using the log4net API:

```
var log = LogManager.GetLogger(typeof(Bar));
log.Debug("Hello World");
```

Additional log information can be attached to a log message by using a range of different contextual options:

```
GlobalContext.Properties["App"] = "MyCoolApp";
using(ThreadContext.Stacks["NDC"].Push("context"))
{
    log.Debug("Hello World");
}
```

You can embed information directly in log messages with log4net, but semantic logging isn't supported.

NLog

While log4net quickly became the default choice, alternatives began to show up. Probably the first real competitor to log4net's dominance was NLog. Originally developed by Jarek Kowalski and with pull requests from almost 100 people, NLog is a great alternative. While log4net pretty much stood still from 2006, NLog just kept going. While Jarek seemed to pull the plug when starting at Google, the community seemed to step up and new releases are still flowing.

Like log4net, NLog contains multiple log targets and is able to log messages to various data stores. NLog can be configured through both XML and C#:

```xml
<nlog>
  <targets>
    <target xsi:type="File"
            name="file"
            fileName="log-file.log"
            layout="${longdate} ${level} ${message}" />
  </targets>

  <rules>
    <logger name="*" minlevel="Trace" writeTo="file" />
  </rules>
</nlog>
```

NLog provides an API similar to log4net for logging messages to the configured set of targets:

```csharp
var logger = NLog.LogManager.GetCurrentClassLogger();
logger.Debug("Hello World");
```

Additional properties can be attached as well:

```csharp
var msg = new LogEventInfo(LogLevel.Debug, "", "Hi");
msg.Properties.Add("App", "MyCoolApp");
logger.Debug(msg);
```

The recent version of NLog supports semantic logging. To add semantic to your log messages, use the curly brace syntax:

```csharp
logger.Info("{OrderId} from {User}", 42, "Kenny");
```

Serilog

In 2013, the .NET logging framework was taken by storm once again. When everyone else was logging text messages, Nicholas Blumhardt introduced something new: structured logging. Unlike existing logging frameworks, Serilog introduced .NET developers to structured event data rather than simple text messages. Serilog quickly became one of the fastest moving projects in the .NET community and remains so today. I really recommend you look into Serilog for logging messages from your .NET apps.

Serilog is typically configured through C# (XML supported as well):

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.File("log-file.txt")
    .CreateLogger();
```

To log messages through Serilog, use the simple API:

```
Log.Debug("Hello World");
```

Serilog has multiple ways to attach custom properties to log messages. Using a similar approach to log4net, you can push properties using a context object:

```
using (LogContext.PushProperty("App", "MyCoolApp"))
{
    Log.Debug("Hello World");
}
```

An enrichment feature can facilitate all log messages using a bit of C#:

```
Logger.Log = new LoggerConfiguration()
    .Enrich.WithEnvironmentUserName()
    .WriteTo.File("log-file.txt")
    .CreateLogger();
```

Semantic logging is an essential part of Serilog too:

```
Log.Information("{OrderId} from {user}", 42, "Kenny");
```

Microsoft.Extensions.Logging

As part of ASP.NET Core, Microsoft released a new logging framework called Microsoft.Extensions.Logging. This new logging framework is available not only for web applications, but basically everything running on the .NET framework. Microsoft.Extensions.Logging acts as both a logging framework as well as an abstraction on top of existing frameworks like NLog and Serilog.

Microsoft.Extensions.Logging collects a lot of experiences from both open source logging frameworks as well as Microsoft's previous attempts (like Logging Application Block). In fact, Microsoft.Extensions.Logging is open source and available on GitHub within the aspnet organization.

Logging is configured through either JSON or C#:

```
var logger = new LoggerFactory()
    .AddFile("log-file.txt")
    .CreateLogger();
```

Logging messages uses an API similar to other logging frameworks:

```
logger.LogDebug("Hello World");
```

Custom properties can be added by using the scope feature of Microsoft.Extensions.Logging. A scope is used to group similar log messages together:

```
using (logger.BeginScope(new[]
    {
        new KeyValuePair<string, object>(
            "App", "MyCoolApp")
    }))
{
   logger.LogDebug("Hello World");
}
```

Semantic logging is supported through the curly brace syntax. Remember that Microsoft.Extensions.Logging can act as a logging abstraction, making it a proxy. If you configure Microsoft.Extensions.Logging to use a logging framework that doesn't support semantic logging, log messages are simply outputted as strings.

```
logger.LogInformation(
    "{OrderId} from {user}",
    42,
    "Kenny");
```

Best Practices

- Implement logging in all projects. You might think that a service or application is too small to benefit from logging, but you always end up needing it at some point.
- Think about which severity to put on each message. I've seen a lot of code where the chosen severity looked totally random. The severities are there for a reason and you will get a much better overview of the log when using the right severity for every single message.
- Don't wrap the logging framework in a transparent proxy. I've seen this implemented multiple times. Someone tries to make it transparent which logging framework is used and therefore creates a custom class for implementing logging. If you really need this, use something like Microsoft.Extensions.Logging or Common Logging.

Logging From Web Applications

Now that you've learned about logging in general and the range of options, I want to switch focus to web applications. At elmah.io, we are developing a range of different web applications, which is why this application type is closest to our hearts. Setting up logging from other types of applications is fully detailed on the documentation for each logging framework.

While being able to log debug and information messages from a web application is important, I want to focus on error logging in this chapter. Monitoring your web application for errors requires much more than being able to tell your boss that all stacktraces are in a log file somewhere.

Website errors divide into two different categories: handled and unhandled. Handled errors are the errors logged by your code. This would typically be a message logged through a logging framework from a catch block. Errors like this can result in an error page for the user, but since you are in control here, it's really up to you to decide how a caught error should be presented to the user. Unhandled are more critical, since they are exceptions with permission to travel all the way from your backend code to the user. Errors like this should result in a nice error page for your users. In our experience, this is far from the reality. You've probably seen a yellow screen of death from a production website recently, right?

You're smart and I bet you already managed to set up logging of handled errors in your code. For the rest of this chapter, I'll focus on the unhandled errors.

Web Frameworks

ASP.NET Web Forms, Web Pages, and MVC

ASP.NET and friends are the most common choices for most .NET developers. Luckily, the platform provides some great hooks for logging unhandled errors. Writing code to log all uncaught exceptions to your log should be the number one priority on any project.

To start building an error log of all uncaught exceptions in ASP.NET, you can implement a method named `Application_Error` inside your `global.asax.cs` file:

```
void Application_Error(object sender, EventArgs e)
{
    var exception = Server.GetLastError();
    Log.Error(exception);
}
```

The code resolves the last thrown error and logs it through some arbitrary log framework. The code doesn't stop the error from being displayed in the browser, but at least you know that something isn't right.

ASP.NET MVC also supports the `Application_Error method`, which, in my opinion, you should use. I still want to say a few things regarding the additional hooks available in MVC, since you will see people recommending these on StackOverflow and similar websites.

There's a concept called `HandleErrorAttribute` that can be attached to both controllers and actions. The `HandleErrorAttribute` only catches errors inside the controller itself, which is why it is a poor choice for logging all uncaught exceptions happening in your application.

Each MVC controller has a `OnException` method that you can override. Again, basing your logging on this message is a poor choice, since that is only called when an error happens inside the scope of your controller.

ASP.NET Web API

ASP.NET Web API is built on top of ASP.NET, which is why `Application_Error` is fully supported. With that said, there's another option for logging all errors when using this framework. Meet Web API exception logging. The exception logger feature in Web API makes it easy to log all exceptions by implementing the `IExceptionLogger` interface:

```
public class LoggingExceptionLogger : IExceptionLogger
{
    public void Log(ExceptionLoggerContext context)
    {
        Log.Error(
            context.ExceptionContext.Exception,
            "Error");
    }
}
```

Your exception logging class is configured with Web API:

```
GlobalConfiguration
    .Configuration
    .Services
    .Add(typeof(IExceptionLogger),
         new LoggingExceptionLogger());
```

Web API now executes your code every time an exception is thrown.

There is an option to implement error logging in Web API filters as well, but we typically don't recommend this for the same reasons as with MVC's `HandleErrorAttribute`.

ASP.NET Core

If you think that error logging in classic ASP.NET seems a bit manual, you'll love ASP.NET Core. Logging is built in from day one. There are basically two options for logging errors from ASP.NET Core and they both have their benefits.

The first option is to use Microsoft.Extensions.Logging, which is installed as part of the ASP.NET Core template. As long as you make sure to configure a destination for log messages, Microsoft.Extensions.Logging will automatically log all exceptions happening in your code.

ASP.NET Core uses dependency injection to make the logger available. To start logging messages, inject an `ILogger` into your controller:

```
public class MyController : Controller
{
    private readonly ILogger _logger;

    public MyController(ILogger<MyController> logger)
    {
        _logger = logger;
    }
}
```

Logging is now simple using the log methods already discussed in the previous chapter:

```
public IActionResult Get()
{
    _logger.LogInformation("Get just called");
    return Ok();
}
```

When logging from ASP.NET Core, configuration is added through the `WebHostBuilder`, rather than needing to create a new instance of the `LoggerFactory`. In `program.cs`, call the `ConfigureLogging` method:

```
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        ...
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConfiguration(hostingContext
                .Configuration
                .GetSection("Logging"));
            logging.AddConsole();
            logging.AddDebug();
        })
        .UseStartup<Startup>()
        .Build();

    webHost.Run();
}
```

The second option lets you log all uncaught exceptions. While Microsoft.Extensions.Logging logs errors thrown from your controllers, you might want to create custom code to log these kind of errors elsewhere. To help you do that, ASP.NET Core introduces the concept of middleware. Middleware are code components executed as part of the request pipeline in Core. If you have a background in ASP.NET and think this sounds familiar, you're right. Middleware is pretty much HTTP modules as you know it from ASP.NET. The biggest difference between modules and middleware is really how you configure it. Modules are configured in `web.config` and since Core doesn't use the concept of a `web.config`, you configure middleware in C#. If you know Express for Node.js, you will find that configuring middleware heavily borrows a lot of concepts from that.

Let's look at some code:

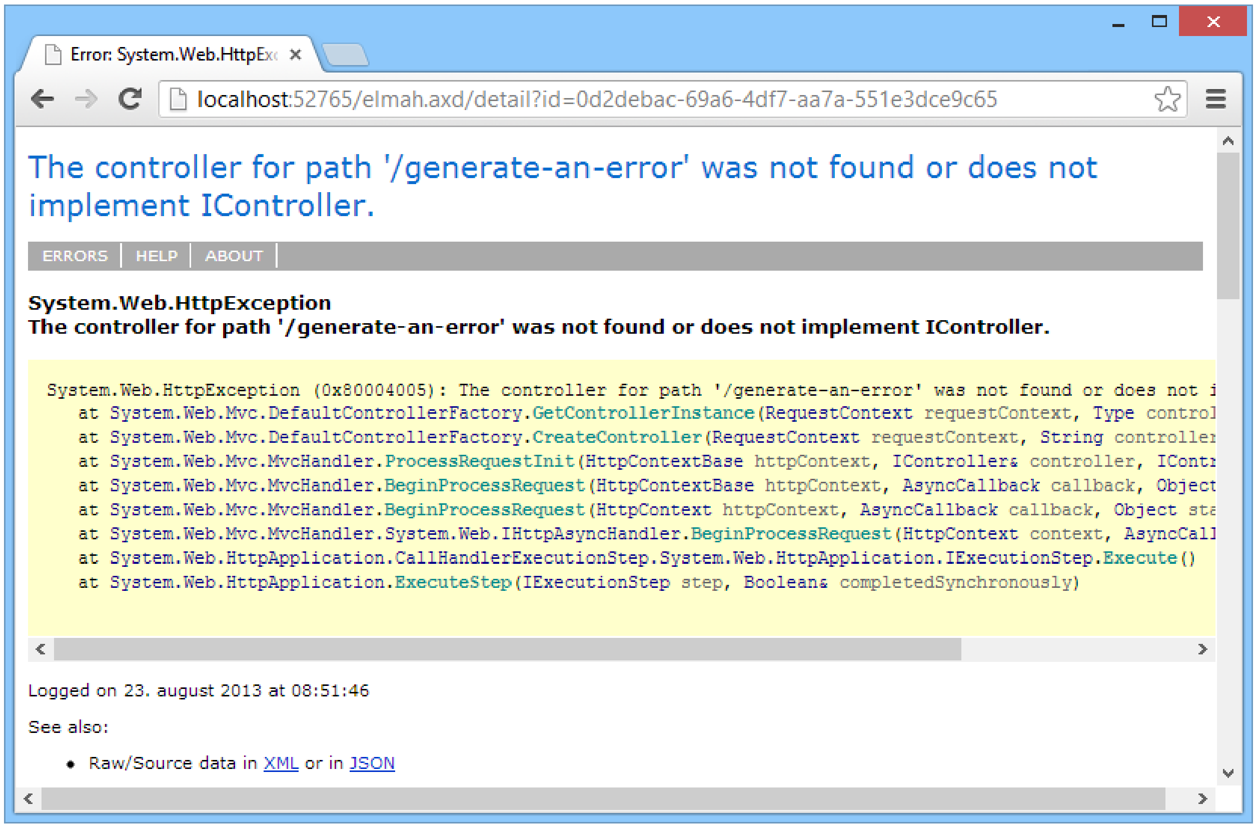
```
public class ErrorLoggingMiddleware
{
    private readonly RequestDelegate _next;

    public ErrorLoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception e)
        {
            Log.Error(e);
            throw;
        }
    }
}
```

In order for our middleware to work, we need to implement two things: a constructor accepting a `RequestDelegate` and an `Invoke` method. The constructor is called a single time, but the underlying delegate will change from request to request. All middleware components are responsible for either executing the next link in the pipeline (`_next`) or terminating the pipeline by not calling `_next`. In our case, we want to execute the rest of the pipeline in order to catch any exceptions happening while processing the HTTP request. When an exception is caught, you can log a message to a logging framework of your choice.

Notice that the catch-block throws the exception after logging it. Rethrowing the exception makes sure that other pieces of middleware handling exceptions still work.

ELMAH

We cannot write about error logging in web applications without mentioning ELMAH. Error logging modules and handlers for ASP.NET (ELMAH) is the de-facto standard error component for .NET. ELMAH has existed for almost a decade but still works as wonderful as when it was initially released. The idea behind ELMAH is to log all uncaught exceptions, including a lot of contextual information about the current HTTP context. ELMAH comes with a simple UI which shows you a list of errors happening and when, as well as some additional information like the type of error, the user causing the error, and more.


Error: System.Web.HttpExc
localhost:52765/elmah.axd/detail?id=0d2debac-69a6-4df7-aa7a-551e3dce9c65
ERRORS
HELP
ABOUT

The controller for path '/generate-an-error' was not found or does not implement IController.

System.Web.HttpException
The controller for path '/generate-an-error' was not found or does not implement IController.

```
System.Web.HttpException (0x80004005): The controller for path '/generate-an-error' was not found or does not i
   at System.Web.Mvc.DefaultControllerFactory.GetControllerInstance(RequestContext requestContext, Type control
   at System.Web.Mvc.DefaultControllerFactory.CreateController(RequestContext requestContext, String controller
   at System.Web.Mvc.MvcHandler.ProcessRequestInit(HttpContextBase httpContext, IController& controller, ICont
   at System.Web.Mvc.MvcHandler.BeginProcessRequest(HttpContextBase httpContext, AsyncCallback callback, Object
   at System.Web.Mvc.MvcHandler.BeginProcessRequest(HttpContext httpContext, AsyncCallback callback, Object sta
   at System.Web.Mvc.MvcHandler.System.Web.IHttpAsyncHandler.BeginProcessRequest(HttpContext context, AsyncCall
   at System.Web.HttpApplication.CallHandlerExecutionStep.System.Web.HttpApplication.IExecutionStep.Execute()
   at System.Web.HttpApplication.ExecuteStep(IExecutionStep step, Boolean& completedSynchronously)
```

Logged on 23. august 2013 at 08:51:46

See also:

- Raw/Source data in XML or in JSON

ELMAH UI

Unfortunately, ELMAH hasn't been ported to ASP.NET Core yet, which is why it only works with classic web apps built on top of frameworks like ASP.NET Web Forms, MVC, and Web API.

Custom Error Pages

A common issue when dealing with logging and web applications is being able to show a nice error page to the user when an exception is thrown, but still log the error. Just by searching StackOverflow and MSDN, this issue alone has cost a lot of developers plenty of hours debugging.

Rather than repeating what I've stated previously, I'd much rather give you a link to what I believe is the ultimate resource on this issue. Dustin Moris Gorski wrote a couple of blog posts which will explain everything you will need to know to set this up correctly:

- Demystifying ASP.NET MVC 5 Error Pages and Error Logging[1]
- Error Handling in ASP.NET Core[2]

[1]https://dusted.codes/demystifying-aspnet-mvc-5-error-pages-and-error-logging
[2]https://dusted.codes/error-handling-in-aspnet-core

Best Practices

- Don't wrap all web methods in try-catch and log exceptions happening. A good web framework allows you to log all uncaught exceptions, as well as show good error pages on exceptions.
- Make sure to choose a framework that logs contextual information when logging from a web application (like ELMAH). Web specific information, like server variables, can be essential in order to debug errors.
- Remember to log all website errors, even when showing a custom error page to the user.

Logging to the Cloud

When talking about logging, I need to mention the cloud. Where it was quite normal to log to local files 5-10 years ago, logging has (like everything else) moved to the cloud. When talking about cloud logging, it's important to be able to distinguish between a couple of different terms.

Cloud Logging vs. Cloud Error Management

I often talk to people that don't realize that there's a big difference between cloud logging and cloud error management or crash monitoring. A lot of cloud-based solutions offering cloud logging through log4net, NLog, and Serilog exist out there. What these solutions have in common is that they collect, index, and make your log messages available through some kind of a web UI. Products like Logentries, Splunk, and Humio are great for collecting massive amounts of messages and making them searchable. These solutions offer great flexibility in terms of creating custom dashboards and much more. Cloud logging platforms are typically used by large organizations since they have greater needs for customizations.

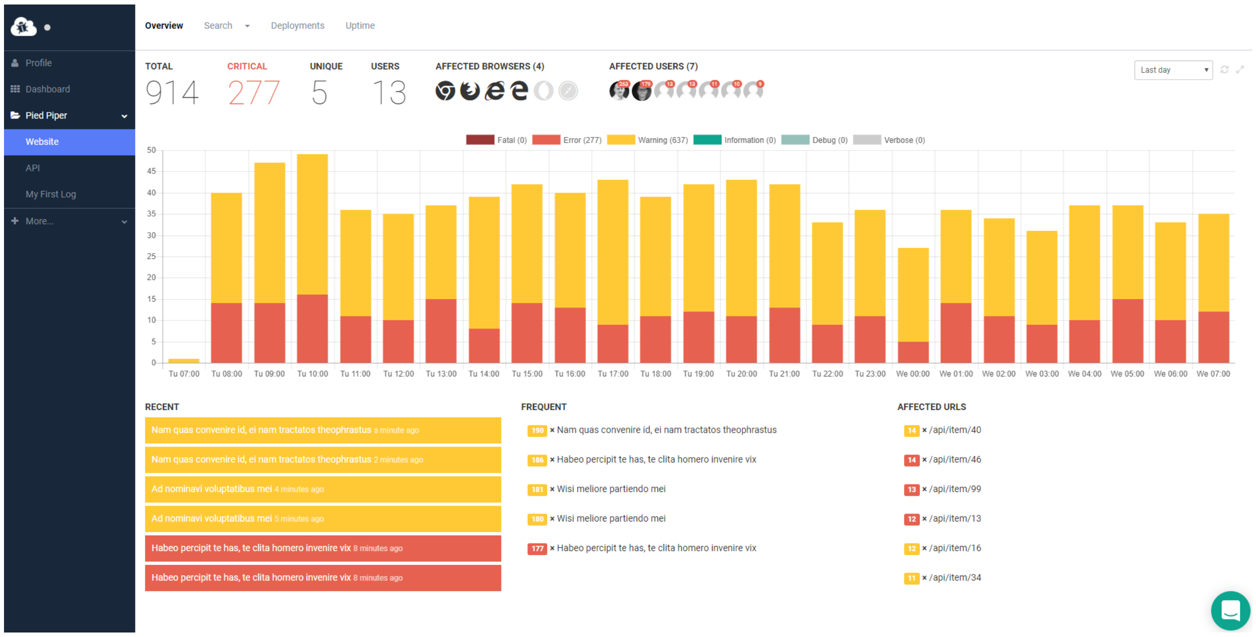
While cloud logging platforms are a great choice for logging text-based or structured log messages to the cloud, they typically don't provide the feature of error management. Having a system optimized for monitoring and handling errors is an essential part of web application development today. Products like elmah.io, New Relic, Raygun, and Stackify are all examples of popular choices for implementing error management. All run in the cloud and provide easy integration with your .NET websites. When looking through the different solutions, some basic features work pretty much the same way, while others vary by product. Some focus on supporting a lot of different programming languages, others on integrating with logging frameworks, and some on integrating with third party systems like Slack and GitHub. What these products have in common is that they help collect errors (and all the detailed information around each error) to help the user see when their website starts failing.

elmah.io

Yet another ELMAH product? Not quite. elmah.io[3] is a cloud-based error management service that I founded 5 years ago. When elmah.io was introduced back in 2013, the intention was to create a cloud-based error logger for ELMAH. We had some simple search and graphing possibilities, but the platform was meant as an alternative to host your own error logs in SQL Server or similar databases.

[3] https://elmah.io/

In time, elmah.io grew from being a hobby project to an actual company. During those years, we realized that the potential of the platform exceeded the possibilities with ELMAH in many ways. New features not available in ELMAH have been added constantly to elmah,io, a process that would have been nearly impossible with ELMAH's many storage integrations.



RECENT

- Nam quas convenire id, ei nam tractatos theophrastus a minute ago
- Nam quas convenire id, ei nam tractatos theophrastus 2 minutes ago
- Ad nominavi voluptatibus mei 4 minutes ago
- Ad nominavi voluptatibus mei 5 minutes ago
- Habeo percipit te has, te clita homero invenire vix 8 minutes ago
- Habeo percipit te has, te clita homero invenire vix 8 minutes ago

FREQUENT

- 190 × Nam quas convenire id, ei nam tractatos theophrastus
- 186 × Habeo percipit te has, te clita homero invenire vix
- 181 × Wisi meliore partiendo mei
- 180 × Wisi meliore partiendo mei
- 177 × Habeo percipit te has, te clita homero invenire vix

AFFECTED URLS

- 14 × /api/item/40
- 14 × /api/item/46
- 13 × /api/item/99
- 12 × /api/item/13
- 12 × /api/item/16
- 11 × /api/item/34

elmah.io Overview

Today, elmah.io is a full error management system for everything from console applications to web apps and serverless code hosted on Azure or AWS. We've built an entire uptime monitoring system able to monitor not only if your website fails, but also if it even responds to requests. I very much encourage you to try out elmah.io to help with implementing error management in your organization.

Best Practices

- Implement a process for error management in your organization. Logging errors isn't enough. You need to follow up on errors happening and fix them fast.
- Choose an error management system first. When you're happy with your choice, start thinking about how to store everything else besides errors.