



TECHNICAL TUTORIAL

Smarter Errors & Logs: Putting the data to work.

Jason Taylor, CTO

Abstract

Have you ever had to work with your log files once your application left development? If so, you quickly run into a few pain points. Learn how to create a “culture of logging” and work smarter not harder

Logging. We should be doing this better by now.

What do I mean? There are lots of logging frameworks and libraries out there, and most developers use one or more of them every day. A few examples off the top of my head for the .Net developers: log4net, nLog, elmah, and the Enterprise Library Logging Application Block. They are simple and easy to use, and work great for developers debugging code. It's still just not enough though.

Have you ever had to work with your log files once your application left development? If so, you quickly run into a few pain points:

- There's a lot more data
- You have to get access to the data
- It's spread across multiple servers
- A specific operation may be spread across service boundaries – so even more logs to dig through
- It's flat and hard to query – even if you do put it in SQL, you are going to have to do a lot of indexing to make it usable
- It's hard to read
- You generally don't have any context of the user, etc
- You probably lack some details that would be helpful (you mean “log.Info(‘In the method’)” isn't helpful???)
- Managing log file rotation and retention

Additionally, you have all this rich data about your app that is being generated and you simply aren't *proactively putting it to work*.

It's time to get serious.

Once you're working on an application that is not in the development environment, logging messages (including exceptions) are usually your only lifeline to *quickly* discovering why something in your app isn't working correctly. Sure, APM tools can alert you to memory leaks and performance bottlenecks, but generally lack enough detail to help you solve a specific problem, i.e. (why can't *this* user log in, or why isn't *this* record processing?).

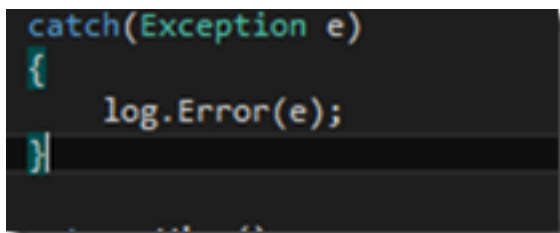
At Stackify, we've built a "culture of logging" which set out to accomplish these goals:

1. *Log all the things*. Log as much as we possibly can, to always have relevant, contextual logs that don't add overhead.
2. *Work smarter, not harder*. Consolidate and aggregate all of our logging to a central location, available to all devs, and *easy to distill*. Also, to find new ways for our logging and exception data to help us *proactively* improve our product.

In this post, we'll explore these concepts, and share what we've done to address it, much of which has become a part of Stackify's Smart Error & Log Management (SmartELM) product.

LOG ALL THE THINGS.

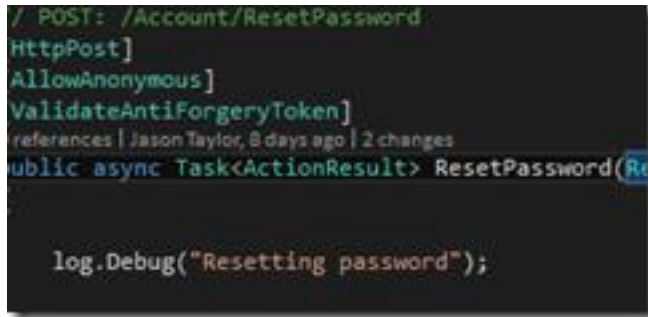
I've worked in a lot of shops where log messages looked like this:



```
catch(Exception e)
{
    log.Error(e);
}
```

I'll give the developer credit: at least they are using a try / catch and handling the exception. The exception will likely have a stack trace so I know where it came from, but no other context. More on this later.

Sometimes, they even do some proactive logging, like this:



```
/ POST: /Account/ResetPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
references | Jason Taylor, 8 days ago | 2 changes
public async Task<ActionResult> ResetPassword(R...

log.Debug("Resetting password");
```

But generally, statements like that don't go a long way towards letting you know what's really happening in your app. If you're tasked with troubleshooting an error in production, and/or is happening for just one (or a subset) of the application users, this doesn't leave you with a lot to go on, especially when considering your log statement could be a needle in a haystack in an app with lots of use.

As I mentioned earlier, logging is often one of the few lifelines you have in production environments where you can't physically attach and debug. You want to log as much relevant, contextual data as you can. Here are our guiding principles on doing that.

Walk the code.

Say you have a critical process that you want to add logging (auditing, really) around so that you can look at what happened. You *could* just put a try / catch around the entire thing and handle the exceptions (which you should) but it doesn't tell you much about what went *into* the request. Take a look at the following, oversimplified example.

Take the following object, which is an Entity Framework Code First POCO object.

```
1 public class Foo
2 {
3     public int ID { get; set; }
4     public int RequiredInt { get; set; }
```

```
5     public string FooString { get; set; }
6 }
```

Notice that it has a field called `RequiredInt` which is an `int` that is, surprisingly, not nullable.

Take the following method, which creates a `Foo` and saves it to the database. Note how I've opened the door for error – the method takes a nullable `int` as an input parameter. I cast it to type `int`, but don't check for null. This could easily cause an exception.

```
1 public static void CreateFoos(int? requiredInt, string fooString)
2 {
3     using (var context = new ApplicationDbContext())
4     {
5         var foo = new Foo();
6         foo.RequiredInt = (int) requiredInt;
7         foo.FooString = fooString;
8
9         context.Foos.Add(foo);
10        context.SaveChanges();
11    }
12 }
13
```

It's an oversimplified scenario, but it serves the purpose well. Assuming this is a really critical aspect of my app (can't have any failed Foos!) let's add some logging so we know what's going on.

```
1  try
2  {
3      log.Debug("Creating a foo");
4
5      using (var context = new ApplicationDbContext())
6      {
7          var foo = new Foo();
8          foo.RequiredInt = (int) requiredInt;
9          foo.FooString = fooString;
10
11          context.Foos.Add(foo);
12          context.SaveChanges();
13      }
14  }
15  catch (Exception ex)
16  {
17      log.Error(ex);
18  }
```

Now, let's create some foos; one that is valid and one that is not:

```
1 CreateFoos(1,"The foo is 1");  
2 CreateFoos(null, "The foo is null");
```

And now we get some logging, and it looks like this:

```
1 DEBUG2014-10-31 13:11:08.9834 [11] Creating a foo [CID:(null)]  
2 DEBUG2014-10-31 13:11:10.8458 [11] Creating a foo [CID:(null)]  
3 ERROR2014-10-31 13:11:10.8673 [11]  
4 System.InvalidOperationException: Nullable object must have a  
5 value.  
6 at  
7 System.ThrowHelper.ThrowInvalidOperationException(ExceptionResource  
8 resource)  
9 at System.Nullable`1.get_Value()  
10 at  
11 HelloStackify.Web.Controllers.GimmeErrorsController.CreateFoos(Null  
12 able`1 requiredInt, String fooString) in  
13 c:\Github\StackifySandbox.Net\HelloStackify.Web\HelloStackify.Web\C  
14 ontrollers\GimmeErrorsController.cs:line 57 [CID:(null)]
```

Now we have some logging – we know when Foos are created, and when they fail to create. But I still feel as if I'm missing some context. Context: let's talk about that. In this example, using log4net, we have a few options.

If you look at the signature for the log4net logging methods (log.Debug, log.Info, etc) you'll see there are a couple of overloaded constructors:

```
1 public static void Debug(this ILog log, string message, object
2   debugData)
3   {
4       log.Debug((object) Extensions.GetMessage(message, debugData));
5   }
```

See that 'object debugData?' Looks like just the ticket, doesn't it? Let's change our code to this:

```
1 using (var context = new ApplicationDbContext())
2 {
3     var foo = new Foo();
4     foo.RequiredInt = (int) requiredInt;
5     foo.FooString = fooString;
6
7     log.Debug("Creating a foo: ",foo);
8
9     context.Foos.Add(foo);
10    context.SaveChanges();
11
12 }
```


And look at the output of our file logging:

```
1 DEBUG2014-11-01 09:26:51.8667 [12] Logging a foo [CID:(null)]
```

Hmm. How about that..... no object. As it turns out, you need to have an appender that supports serializing that object and outputting it to wherever it logs (in this case, a file appender). So, we have a couple of additional steps to perform.

1. Serialize the object ourselves (Json.Net works great for this)
2. Create a logging string with a string builder / formatter

```
1 log.Debug(string.Format("Creating a foo:
  {0}", JsonConvert.SerializeObject(foo)));
```

This will produce some log output that looks like this

```
1 DEBUG2014-11-01 10:39:53.3295 [11] Creating a foo:
  {"ID":0,"RequiredInt":1,"FooString":"The foo is 1"}
```

and while it serves the purpose, there are some major drawbacks, mainly in that it's all one string and the more objects and data you add, the more code you are writing. I don't know about you, but when I'm writing log statements, I want it to be fast and easy. The Stackify appenders support logging objects without needing to do any serialization first. It makes it really easy to log complex dynamic objects such as this:

```
1 using (var context = new ApplicationDbContext())
2 {
3     var foo = new Foo();
4     foo.RequiredInt = (int) requiredInt;
5 }
```

```
6 context.Foos.Add(foo);  
7 context.SaveChanges();  
8 log.Debug("Created a Foo", new {  
9     Foo=foo, CreatedBy=User.Identity.Name});  
}
```

Which can yield great output like this:



The screenshot shows a log entry from the Stackify Logging Dashboard. The log message is "11:17:20.480 JTSURFACE DEBUG Created a Foo". To the right of the message is a JSON object representing the context data. The JSON object has a "json" property which is an object containing a "_Foo" property (an object with "id", "fooString", and "requiredInt" properties) and a "createdBy" property with the value "jttaylor@stackify.com".

```
11:17:20.480 JTSURFACE DEBUG Created a Foo {"_Foo":{"id":"16","fooSt...  
"json":  
{  
  "_Foo": {  
    "id": "16",  
    "fooString": "The foo is 1",  
    "requiredInt": "1"  
  },  
  "createdBy": "jttaylor@stackify.com"  
}
```

(This output came from the Stackify Logging Dashboard, via the Stackify log4net appender. More on that later).

Diagnostic context logging

And this brings us to one last point: diagnostic context logging. You'll notice that I logged the user that created the object. When it comes to debugging a production issue, you might have the "Created a Foo" message thousands of times in your logs, but with no clue who the logged in user was that created it. This is the sort of context that is priceless in being able to quickly resolve an issue. Think about what other detail might be useful – for example, `HttpRequest` details. But who wants to have to remember to log it every time? Diagnostic context logging to the rescue. Log4net, for example, makes this really easy. (You can read about the `LogicalThreadContext` here:

<http://logging.apache.org/log4net/release/sdk/log4net.LogicalThreadContext.html>)

To enable diagnostic context logging is really easy. In your log4net config, set up your logical thread context variables:

```
1 <appender name="StackifyAppender"  
2   type="StackifyLib.log4net.StackifyAppender, StackifyLib.log4net">  
3   <logicalThreadContextKeys>User,Request</logicalThreadContextKeys>  
   </appender>
```

Then, just log your context items. In my example, I'm logging data from the User (IPrincipal) and Request (HttpRequest) objects. The simplest way to do this is in my Global.asax on each request.

```
1 void MvcApplication_AuthenticateRequest(object sender, EventArgs  
2   e)  
3   {  
4       try  
5       {  
6           log4net.LogicalThreadContext.Properties["User"] = User;  
7       }  
8       catch (Exception ex)  
9       {  
10          log.Error(ex);  
11      }  
12  }
```

```
1 } void MvcApplication_BeginRequest(object sender, EventArgs e)
1 {
1
2
1 log4net.LogicalThreadContext.Properties["Request"] = new
3 {
1     HostAddress = Request.UserHostAddress,
4     RawUrl = Request.RawUrl,
1     QueryString = Request.QueryString,
5     FormValues = Request.Form
1
6 };
1 }
7
```

(Note: because I don't want the *entire* Request object, I just created a dynamic object with the properties that I care about)

Now, I can simplify my logging statement back to something like this:

```
1 log.Debug("Created a Foo", foo);
```

And get *beautiful* logging statements output like so:

```
11:29:51.628 JTSURFACE DEBUG Created a Foo {"id":"19","requiredInt":1,"fooString":"The foo is 1","_context":{"_request":{"rawUrl":"/GimmeErrors","hostAddress":"::1","FormValues":[],"QueryString":[]},"_user":{"_identities":{"nameClaimType":"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name","isAuthenticated":"True","Claims":{
```

This brings us to the next topic, which is Work Harder, Not Smarter. But before that, I'm going to address a question I'm sure to hear a lot of in the comments: "But if I log *everything* won't that create overhead, unnecessary chatter, and huge log files?" My answer comes in a couple of parts: first, use the logging verbosity levels. you can `log.Debug()` **everything you think you'll need**, and then set your config for production appropriately, i.e. Warning and above only. When you do need the debug info, it's only changing a config file and not redeploying code. Second, if you're logging in an *async, non-blocking way* (such as the Stackify appender), then overhead should be low. Last, if you're worried about space and log file rotation, there are smarter ways to do it, and we'll talk about that in the next section.

Work Smarter, Not Harder.

Now that we're logging *everything*, and it's great, contextual data, we're going to look at the next part of the equation. As I've mentioned, and demonstrated, just dumping all of this out to flat files (or even SQL for that matter) still doesn't help you out a lot in a large, complex application and deployment environment. Factor in thousands of requests, files spanning multiple days, weeks, or longer, and across multiple servers, you have to consider how you are going to quickly find the data that you need.

What we all really need is a solution that:

- Aggregates all Log & Exception data to one place

- Makes it available, instantly, to everyone on your team
- Presents a timeline of logging throughout your entire stack / infrastructure
- Is highly indexed, searchable, and “smart” about it.

This is the part where I tell you about Stackify SmartELM. As we sought to improve our own abilities to quickly and efficiently work with our log data, we decided to make it a core part of our product (yes, we use Stackify to monitor Stackify) and share with our customers, since we believe it’s an issue central to application performance and troubleshooting.

First, we realize that lots of developers already have logging in place, and aren’t going to want to take a lot of time to rip that code out and put new code in. That’s why we’ve created logging appenders for a few of the most common frameworks (detailed information on setup and configuration for all can be found [here](#)):

- log4net
- log4j
- nLog
- logback
- Elmah
- Or write directly to our API using our common library, included with all these appenders or by itself.

Additionally, by using one of these libraries, you also get access to Custom Metrics for [.Net](#) and [Java](#) which go along great with our App & Server monitoring product.

Continuing with log4net as a sample, the setup is easy. Just add the binaries to the project (you can just dump them in the \bin folder and no need to recompile) and add in some web.config-fu:

```
1 <log4net>
2   <root>
3     <level value="ALL" />
4     <appender-ref ref="StackifyAppender" />
```

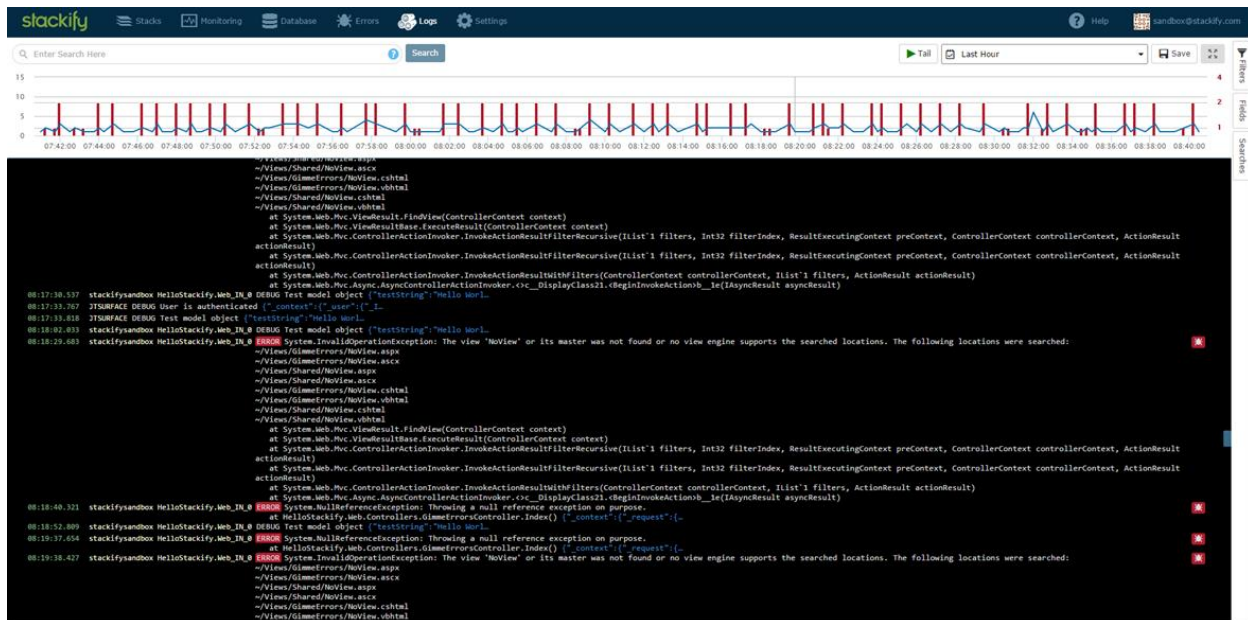
```

5     <appender-ref ref="LogFileAppender" />
6
7 <!--Use the following to send only exception and error statements
8 to Stackify -->
9     <appender name="StackifyAppender"
10    type="StackifyLib.log4net.StackifyAppender, StackifyLib.log4net">
11
12    <logicalThreadContextKeys>User,Request</logicalThreadContextKeys>
13
14    </appender>
15
16    <appender name="LogFileAppender"
17    type="log4net.Appender.RollingFileAppender">
18
19        <param name="File" value="stackify.log" />
20
21        <param name="AppendToFile" value="true" />
22
23        <rollingStyle value="Size" />
24
25        <maxSizeRollBackups value="10" />
26
27        <maximumFileSize value="10MB" />
28
29        <staticLogFileName value="true" />
30
31        <layout type="log4net.Layout.PatternLayout">
32
33            <param name="ConversionPattern" value="%-5p%d{yyyy-MM-dd
34            HH:mm:ss.ffff} [%thread] %m [CID:%property{clientid}]%n" />
35
36            </layout>
37
38        </appender>
39
40    </log4net>

```

As you can see, if you're already using an appender, you can keep it in place and put them side-by-side. Now that you've got your logs streaming to Stackify (by the way, if

our monitoring agent is installed, you can also pipe Windows Event viewer and Syslog files to Stackify as well!) we can take a look at the logging dashboard:



This dashboard shows a consolidated stream of log data, coming from all your servers and apps, presented in a timeline. From here, you can quickly:

- Load logs based on a time range
- Filter for specific server(s) and app(s) or environment(s)

Plus there are a couple of really great usability things built in. One of the first thing you'll notice is that chart at the top. It's a great way to quickly "triage" your application. The blue line indicates the rate of log messages, and the red bars indicate # of exceptions in the log detail. In the screenshot above, the rate of both looks pretty steady, but if I zoom this chart out to the last 7 days it looks like this:



It's abundantly clear that a couple of days ago, my web app started having a lot more activity (as shown by the logging level – Stackify monitoring would corroborate this data

By zooming in on the chart, to this time period, I can quickly filter my log detail down to that time range and take a look at the my logs for that time range.



See that blue text, that looks like a json object? Well, it is a json object. That's the result of logging objects, and adding context properties (also objects) earlier. It looks a lot nicer than plain text in a flat file, doesn't it? Well, it gets even more awesome. See the search box at the top of the page? I can put in any search string that I can think of, and it will query all my logs as if it were a flat file. As we discussed earlier, however, this isn't *great* because you could end up with a lot more matches than you want. Suppose that I want to search for "Hello World" but only in the context of a property called "testString?"

Fortunately, our log aggregator is smart enough to help in this situation. That's because when we find serialized objects in logs, we index each and every field we find. That makes it easy to perform a search like this:

json.testString: "Hello World"

and get just the values that I'm looking for. But I can dig deeper. Say I want to look just for that statement and filter by a specific user (remember, we added the logged in user to every statement via the LogicalThreadContext):

json.testString: "Hello World" AND json._context._user._Identity.name:
jtaylor@stackify.com

That yields the following results, with the search strings / objects specifically highlighted.

Search: `json.testString: "Hello World" AND json._context._user._Identity.name: jtaylor@stackify.com` [Search] [Last Hour] [Save]

Timeline: 08:10:00 08:12:00 08:14:00 08:16:00 08:18:00 08:20:00 08:22:00 08:24:00 08:26:00 08:28:00 08:30:00 08:32:00 08:34:00 08:36:00 08:38:00 08:40:00 08:42:00 08:44:00 08:46:00 08:48:00 08:50:00 08:52:00 08:54:00 08:56:00 08:58:00 09:00:00 09:02:00 09:04:00 09:06:00 09:08:00

Showing 3 records

November 2nd, 2014 (2014-11-02)

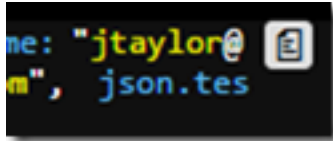
08:17:33.818 3FSURFACE DEBUG Test model object {"testString":"Hello World", "json._context._user._Identity._Claims_value": "jtaylor@stackify.com", "json._context._user._Identities_name": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json._context._user._Claims_value": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json.testString": "Hello World", "json._context._user._Identities._Claims_value": "jtaylor@stackify.com"}

```
{
  "testString": "Hello World",
  "context": {
    "_user": {
      "_Identities": [
        {
          "nameClaimType": "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name",
          "isAuthenticated": "True",
          "claims": [
            {
              "issuer": "LOCAL AUTHORITY",
              "originalIssuer": "LOCAL AUTHORITY",
              "value": "R2d1e5ed-82bf-482e-b222-5862cc29581d",
              "valueType": "http://www.w3.org/2001/XMLSchema:string",
              "type": "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameIdentifier",
              "properties": {}
            }
          ]
        }
      ]
    }
  }
}
```

08:22:00.682 3FSURFACE DEBUG Test model object {"testString":"Hello World", "json._context._user._Identity._Claims_value": "jtaylor@stackify.com", "json._context._user._Identities_name": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json._context._user._Claims_value": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json.testString": "Hello World", "json._context._user._Identities._Claims_value": "jtaylor@stackify.com"}


08:32:07.892 3FSURFACE DEBUG Test model object {"testString":"Hello World", "json._context._user._Identity._Claims_value": "jtaylor@stackify.com", "json._context._user._Identities_name": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json._context._user._Claims_value": "jtaylor@stackify.com", "json._context._user._Claims_Subject_name": "jtaylor@stackify.com", "json.testString": "Hello World", "json._context._user._Identities._Claims_value": "jtaylor@stackify.com"}

Want to know what else you can search by? Just click on the document icon

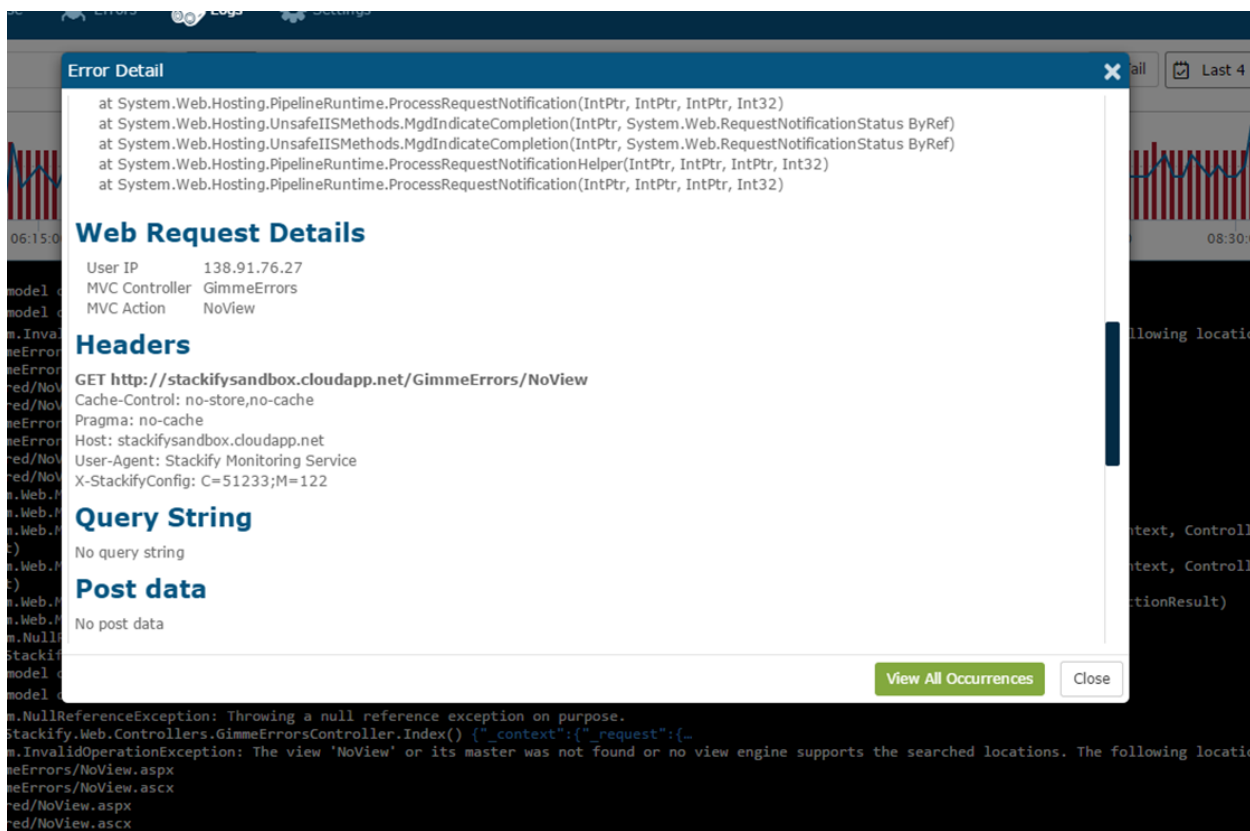


when you hover over a log record, and you'll see all the fields that Stackify indexes. (Note: at the time of this writing, we are preparing to release a new search UI that will provide a dynamic list of filters that you can one-click apply without having to know any query syntax.)

Exceptions

You may have also noticed this little red bug icon  next to exception messages.

That's because we treat exceptions a bit differently. Click on it and we present a deeper view of that exception.

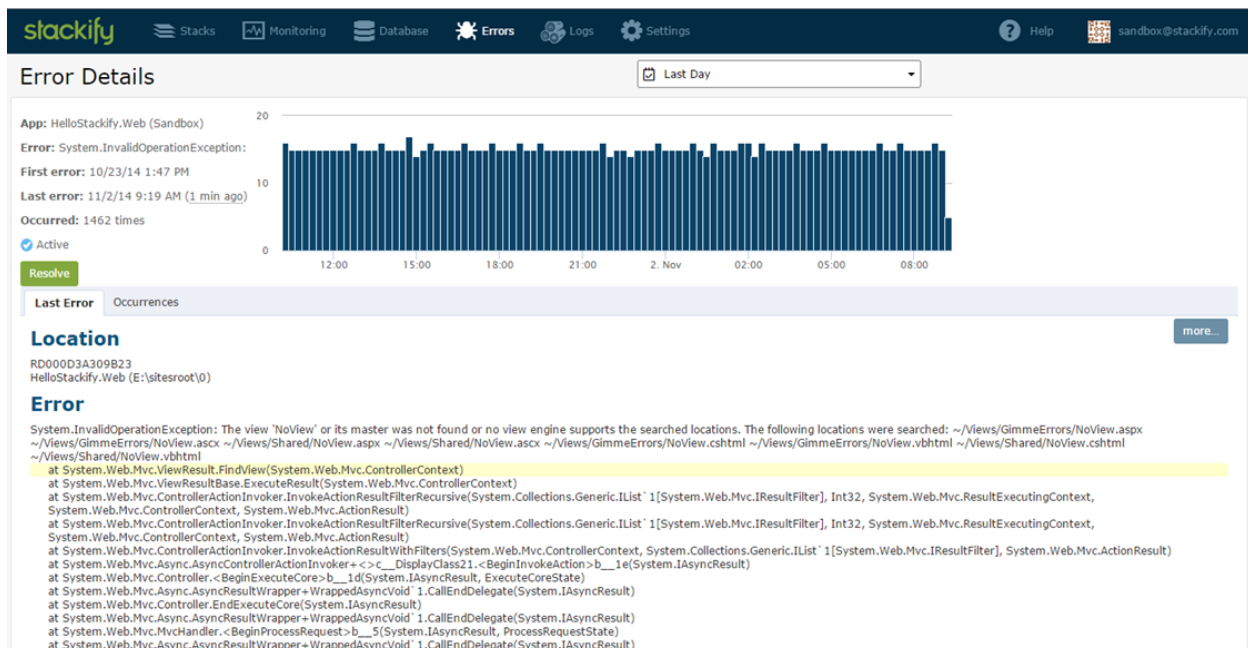


The screenshot shows the 'Error Detail' window in the Stackify application. The window is titled 'Error Detail' and has a close button (X) in the top right corner. It displays the following information:

- Stack Trace:**
 - at System.Web.Hosting.PipelineRuntime.ProcessRequestNotification(IntPtr, IntPtr, IntPtr, Int32)
 - at System.Web.Hosting.UnsafeIISMethods.MgdIndicateCompletion(IntPtr, System.Web.RequestNotificationStatus ByRef)
 - at System.Web.Hosting.UnsafeIISMethods.MgdIndicateCompletion(IntPtr, System.Web.RequestNotificationStatus ByRef)
 - at System.Web.Hosting.PipelineRuntime.ProcessRequestNotificationHelper(IntPtr, IntPtr, IntPtr, Int32)
 - at System.Web.Hosting.PipelineRuntime.ProcessRequestNotification(IntPtr, IntPtr, IntPtr, Int32)
- Web Request Details:**
 - User IP: 138.91.76.27
 - MVC Controller: GimmeErrors
 - MVC Action: NoView
- Headers:**
 - GET http://stackifysandbox.cloudapp.net/GimmeErrors/NoView
 - Cache-Control: no-store,no-cache
 - Pragma: no-cache
 - Host: stackifysandbox.cloudapp.net
 - User-Agent: Stackify Monitoring Service
 - X-StackifyConfig: C=51233;M=122
- Query String:**
 - No query string
- Post data:**
 - No post data

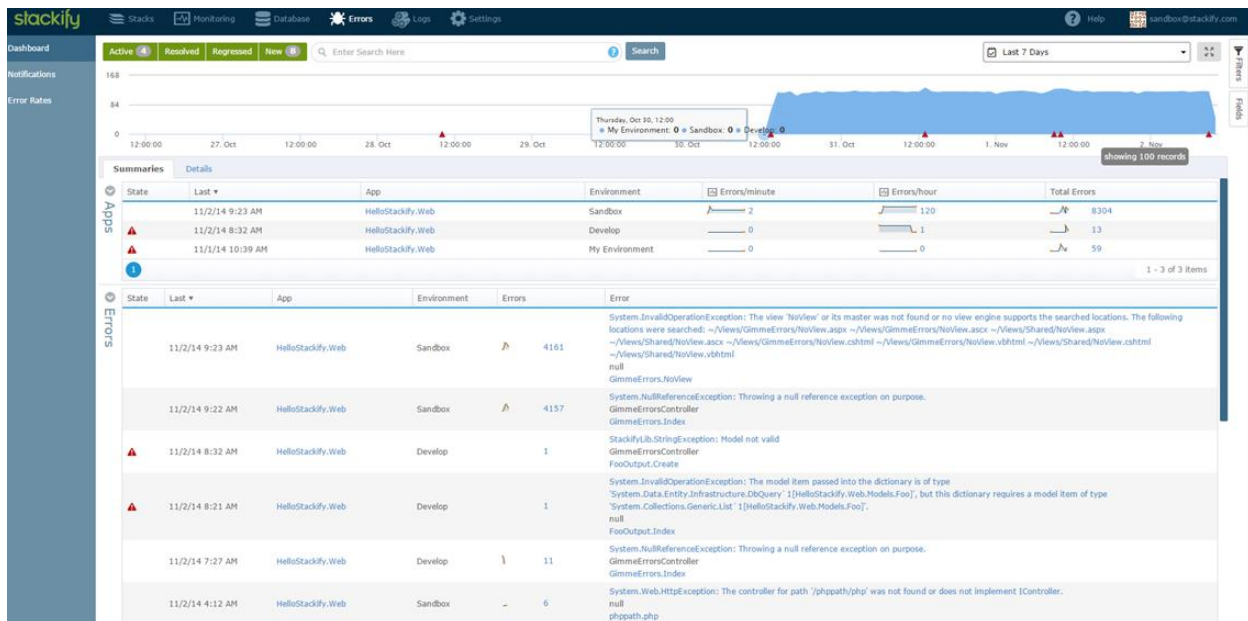
At the bottom of the window, there are two buttons: 'View All Occurrences' and 'Close'. The background of the application shows a log of errors, with the current error being a 'NullReferenceException: Throwing a null reference exception on purpose.' and a 'InvalidOperationException: The view 'NoView' or its master was not found or no view engine supports the searched locations. The following locations were searched: NoView.aspx, NoView.ascx, NoView.aspx, NoView.ascx'.

Our libraries not only grab the full stack trace, but all of the web request details, including headers, querystrings, post data, and server variables. In this modal, there is a “Logs” tab which gives you a pre-filtered view of the logging from the app that threw the error, on the server where it occurred, for a narrow time window before and after the exception, to give more context around the exception. Curious about how common or frequent this error occurs, or want to see details on other occurrences? Click the “View All Occurrences” button and voila!



I can quickly see this error has occurred 1462 times over the last day, and at a pretty steady rate. It tells me, as a developer, that I have a pretty persistent bug.

Errors and Logs are closely related, and in an app where a tremendous amount of logging can occur, exceptions could sometimes get a bit lost. That's why we've built an Errors Dashboard as well, to give you this same consolidated view but limited to exceptions.

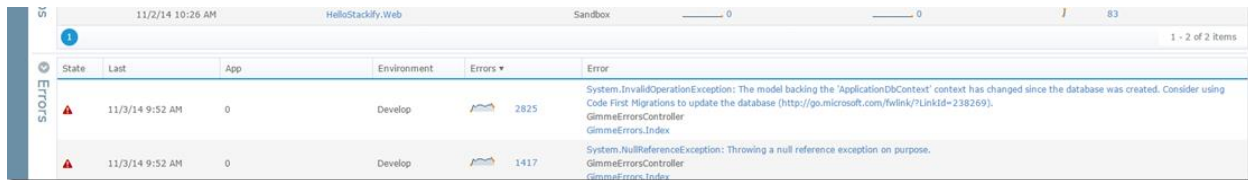


Here I can see a couple of great pieces of data:

- I've had a big uptick in my rate of exceptions over the past couple of days
- The majority of my errors are coming from my "Sandbox" environment – to the tune of about 120 per hour
- I have a couple of new errors that have just started occurring (as indicated by the red triangles)

Have you ever put a new release of your app out to production and wondered what QA missed? (Not that I'm saying QA would ever miss a bug.....) Error Dashboard to the rescue. You can watch real time and see a trend – lots of red triangles, lots of new bugs. Big spike in the graph? Perhaps you have an increase in usage, so a previously known error is being hit more; or perhaps some buggy code (like a leaking SQL connection pool) went out and is causing a higher rate of SQL timeout errors than normal.

In fact, as I was writing this article, I published a change to the app I'm using, and forgot to first publish the EF Code First migrations (and did not have automatic migrations turned on). This scenario is exactly what I'm referring to. Front and center on my dashboard, I see A new error around this, and it's happening a lot.



State	Last	App	Environment	Errors	Error
▲	11/3/14 9:52 AM	0	Develop	2825	System.InvalidOperationException: The model backing the 'ApplicationDbContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269). GimmeErrorsController GimmeErrors.Index
▲	11/3/14 9:52 AM	0	Develop	1417	System.NullReferenceException: Throwing a null reference exception on purpose. GimmeErrorsController GimmeErrors.Index

It's not hard to imagine a lot of different scenarios to which this could provide early warning and detection for. Hmm. Early warning and detection. That brings up another great topic.

Monitor

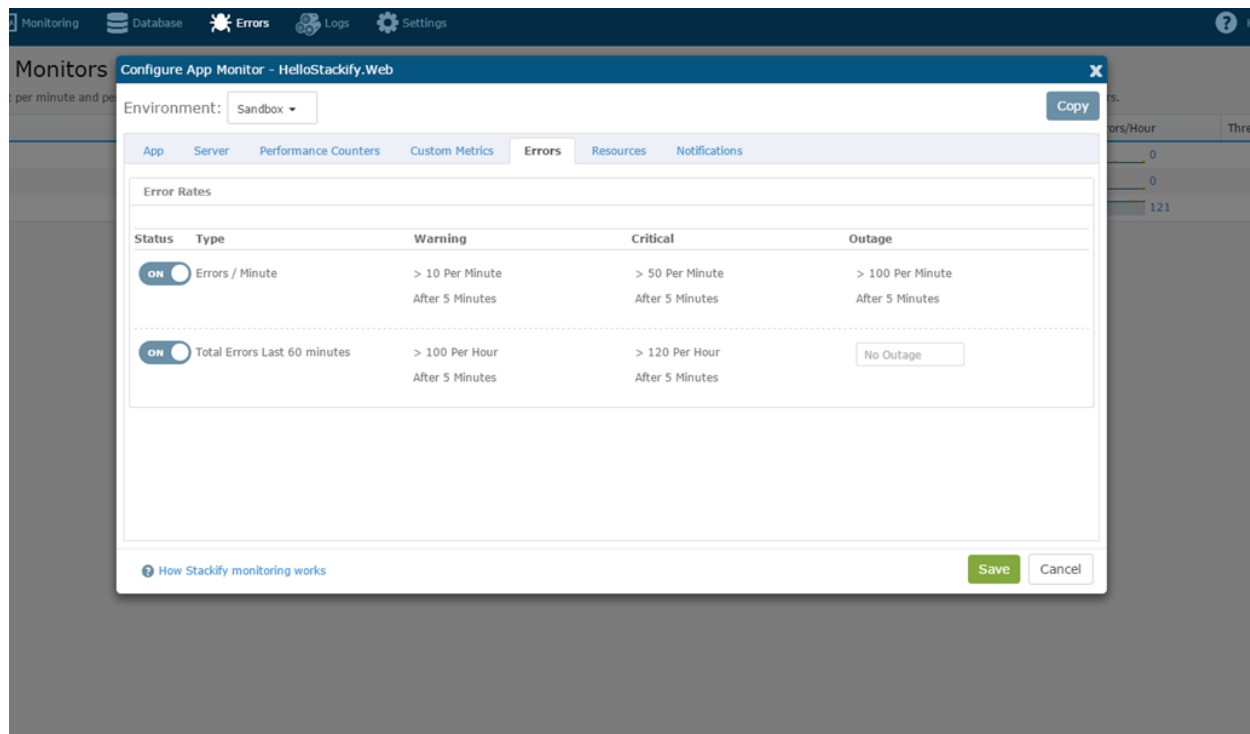
Wouldn't it be nice to be alerted when:

- An error rate for a specific app or environment suddenly increases?
- An error that was specifically resolved starts happening again?
- A certain action that you log does not happen enough, too often, etc?

Stackify can do all of that. Let's take a look at each.

Error Rates

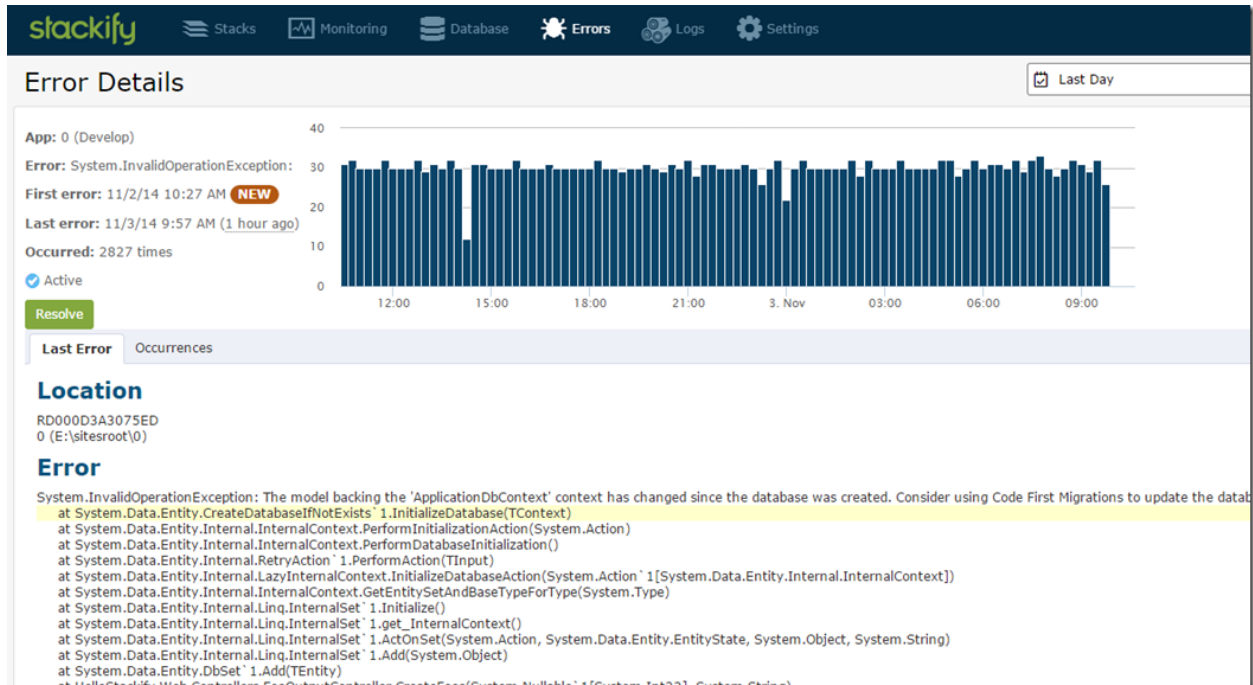
When we looked at the error dashboard, I noted that my 'Sandbox' environment is getting a high number of errors per hour. From the Error dashboard, click on "Error Rates" and then select which app / environment you wish to configure alerts for:



I can configure monitors for “Errors / Minute” and “Total Errors Last 60 minutes” and then choose the “Notifications” tab to specify who should be alerted, and how. Subsequently, if using Stackify Monitoring, I can configure all of my other alerting here as well: App running state, memory usage, performance counters, custom metrics, ping checks, and more.

Resolved Errors & New Errors

Earlier on, I introduced a new error by not publishing my Code First migrations when I deployed my app. I’ve since fixed that and confirmed it by looking at the details for that particular error. As you can see, the last time it happened was 45 minutes ago:



It was a silly mistake to make, but one that is easy to make as well since I don't have automatic migrations enabled. I'm going to mark this one as "resolved" which lets me do something really cool: get an alert if it comes back. The Notifications menu will let me check my configuration, and by default I'm set to receive both new and regressed error notifications for all my apps and environments.

Error Notifications **Add New**

For each notification group configure which environments and apps they should receive emails for when a new email occurs or if a fixed error regresses.

Notification Group	Environments	Apps	New Errors	Regressed Errors
Default Notification Group	All Environments	All Apps	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Now, if the same error occurs again in the future, I'm going to get an email about the regression and it shows up on the dashboard as such. This is great little bit of automation to help out when you "think" you've solved the issue and want to make sure.

Log Monitors

Some things aren't very straightforward to monitor. Perhaps you have a critical process that runs asynchronously and the only record of it's success (or failure) is logging statements. Earlier in this post, I showed the ability to run deep queries against your log data, and any of those queries can be saved, and monitored. I've got a very simple scenario here: my query is executed every minute, and we can monitor how many matched records we have.

ng Database Errors Logs Settings

New

Configure Log Monitor - jtaylor logs hello world

Logs Notifications

Check Frequency
Every Minute

Monitor Name
jtaylor logs hello world

Saved Query
Enter a search query to monitor against all logs or use a saved search to use more advanced filter options. Create saved searches from the main log explorer screen.

Enter a search query
Use a saved search

Saved Search
jtaylor logs hello world

Search Time frame
Last 60 minutes

Search: json.testString: "Hello World" AND json._context._user._Identity.name: jtaylor@stackify.com

Status Type
ON Query Matches

Warning
< 10
After 5 Minutes

Critical
No Critical

Outage
No Outage

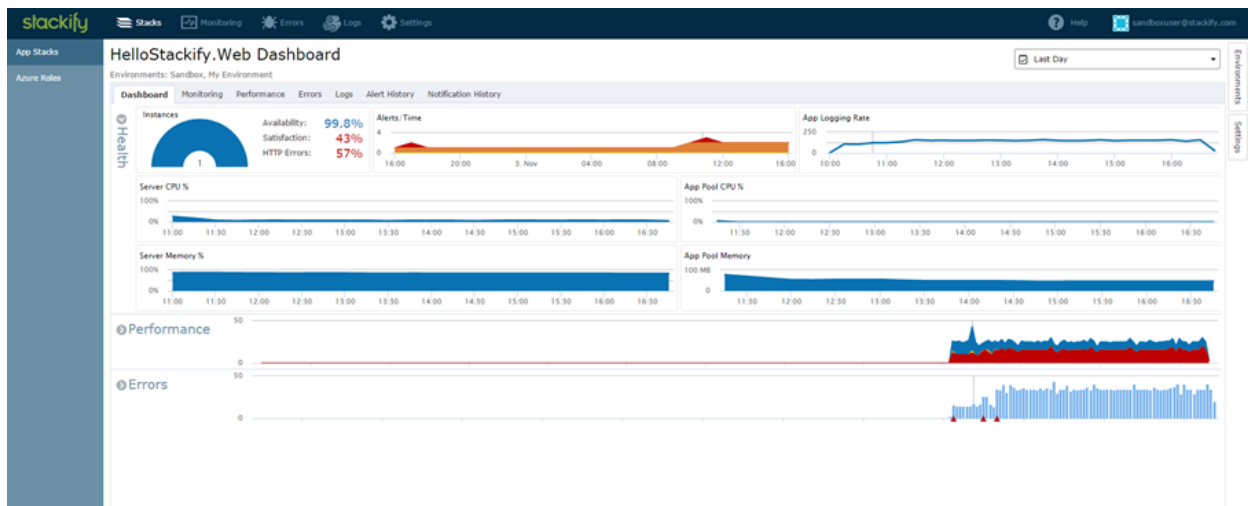
How Stackify monitoring works

Delete Save Cancel

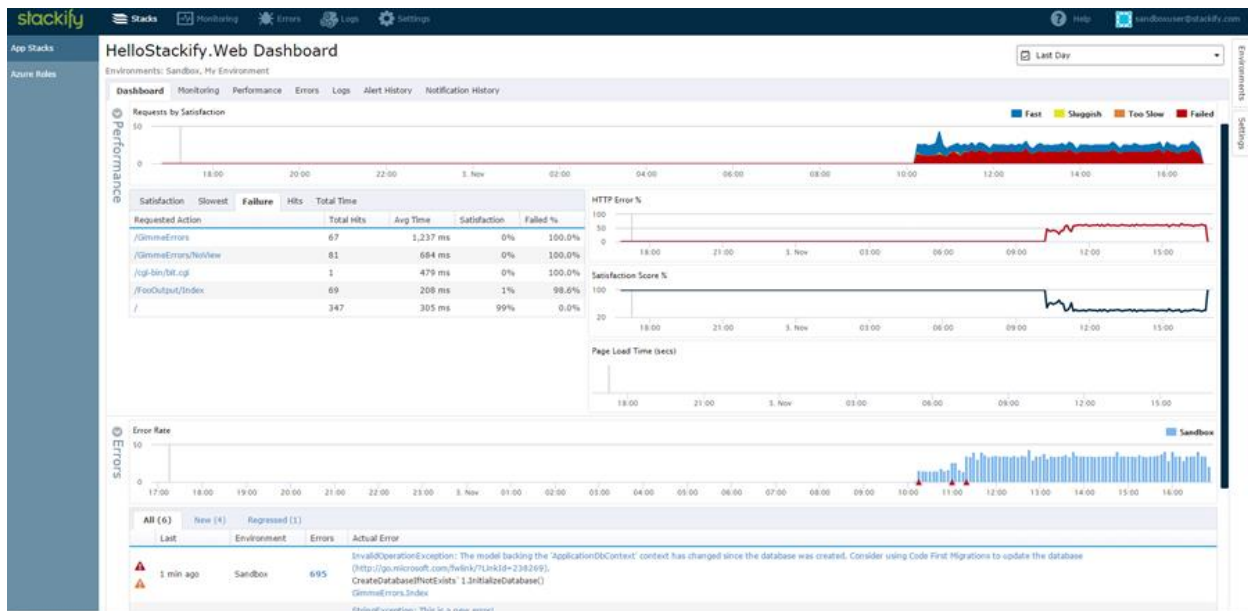
It's just a great simple way to check system health if a log file is your only indication. (Here's a pro-tip: send your Windows Event Logs to Stackify and monitor for W3WP process crashes and other stack overflow type events).

Wrapping Up

All of this error and log data can be invaluable, especially when you take a step back and look at a slightly larger picture. Below is the Application Dashboard for this app that contains all of the monitoring:



As you can see, you get some great contextual data at a glance that errors and logs contribute to: Availability, Satisfaction, and Errors. The site is up (99.8%) but the user satisfaction is low and errors are high. You can quickly start drilling down to see which pages are not performing well, and what errors are occurring:



And when looking at a specific error, see the associated logs.

All said and done, this has helped us tremendously in improving our own product, to quickly detect and resolve application issues.

There was a lot to cover in this post, and I feel like I barely scratched the surface. If you dig a little deeper, or even get your hands on it, you can! I've made the sample app [available here on GitHub](#) and you can log into our demo environment and try out all the features I've demo'd here.

Just go to <http://s1.stackify.com>

Login as: sandboxuser@stackify.com with the password 'sandbox-1' and you will be in a (limited) account where you can view and query all of the errors and logs, plus see the application monitoring.

About Stackify

Stackify is the industry's only platform that combines error-aware log management and smart error aggregation. The solution optimized for dynamic environment, providing cross-server aggregation all errors and logs. Stackify also provide an optional comprehensive application monitoring, performance management, custom metrics, notification, and secured data access for better app troubleshooting. Stackify provides developers and operations with DevOps insight, allowing them to detect issues before they affect business and shorten time to resolution to ensure a better end user experience.

Try it now for [free](http://www.stackify.com/) at <http://www.stackify.com/>