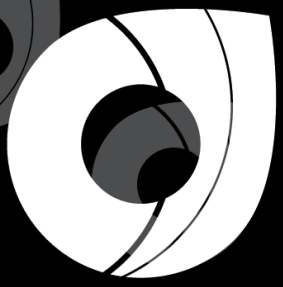
Corndel DevOps Engineering Programme

in association with Softwire

Module 13: Logging, Monitoring & Alerting



Corndel Digital.

DB55

Introduction

Any sufficiently complicated system *will* go wrong. It's important to be aware when this happens, and to have the information needed to fix it when it does. Logging, Monitoring and Alerting together provide visibility to see when a system is working, or not, and invaluable information for fixing problems that arise.

Logging

In the simplest terms, "logs" are records of events that have happened in the system.

This expression comes from the old days of sailing, where the ship's speed would be measured by throwing a log tied to a rope overboard, and the results written down in a "log-book" to keep track of progress.

In our modern context, "logs" consist of a series of messages stored with associated timestamps in "logfiles", and are widely used across the software ecosystem to provide audit trails and data to help understand the activity of complex systems and diagnose errors.

Monitoring

Monitoring is about the status of a system at any given moment in time. It consists of metrics like CPU usage and requests per minute, as well as binary checks like "is my application actually running?". As well as detecting problems at this moment in time, a history of these values is useful for diagnosing and predicting problems.

Alerting

Alerting brings logging and monitoring together. Alerting rules can be configured to send emails, text messages or other notifications when particular events are logged, or metrics hit thresholds. These enable problems to be fixed proactively rather than waiting for them to be raised by users.

Table of Contents

Core Material

Additional Material

Chapter 1

Application Logs

Lots of different systems generate logs. Web servers, databases, operating systems, etc. If it runs on a computer, it is likely to be recording logs somewhere. **Application logs** are those logs generated by a specific program, generally the one you are responsible for supporting.

In this chapter we will discuss why we need application logs; how we might go about creating things; and finally how we manage those logs over time.

Why do we need logs?

To debug live services

Often when a crash or error occurs, the first steps are to discover what was happening in the system to cause it. Having well-thought-out logs can be invaluable in spotting where two services failed to interact, or what exact steps the user was taking prior to the failure. If this data is not being constantly outputted and stored whilst the software is running, it can take much longer to diagnose and fix issues that arise.

To provide summaries of app usage

There is a lot of value in knowing how your users are actually using the application you have provided them. By keeping track of every action each user takes, and which features are most used, this data can be utilised to aid in future planning and optimisations. This is called 'Analytics', and beyond logging there is a whole industry decided to tools to collect and analyse this data.

To monitor for malicious activity

Simply logging any connections to the most secure parts of the system can provide a good audit trail and deterrent to prevent malicious activity. For instance, any direct access to a certain database using employee credentials could be investigated and the justification confirmed.

It is also common on larger systems to set up so-called "honey pots" to attract attackers. These are fake resources that appear to be sensitive, but are never connected to through normal usage of the application, so can be used to track attempted intrusions.

Creating application logs

Setting up a logger

Although almost every language has a variety of simple methods of outputting data or writing messages to a file on disk, it is strongly recommended to use a specific logging package for the purpose of logging.

These packages handle common issues and pitfalls, including ensuring the logs are consistently formatted, saving different types of logs for different environments, preventing concurrent writing to the log-files, and rotating log files to save on disk space.

In some languages, it is common practise to create a new instance of the Logger for each class or subsection of the application, which allows easy labelling of the source of the message in the outputted logs.

Ensuring only useful data is logged

Whilst setting up and writing log messages, keep in mind what the end purpose is.

- Cryptic messages that rely on a deep understanding of the internals of a system are not going to be useful in an emergency diagnosis of errors.
- Audit trails should contain the required information and user-ids, in a format that is readable to both humans and software-based parsers.
- Enormous log files can be extremely difficult to work with when looking for a specific series of events, and take up unnecessary space on disk, so be mindful of which events are logged.
- Sensitive information needs to be appropriately redacted, or simply never logged. Including passwords or confidential user information in logs creates a new attack surface that hackers could exploit, and also means that logs can't be as freely distributed to developers who don't have live access.

Log levels

Not all events that take place across an application are equal. Simple button clicks may be useful when debugging a problem locally, but logging them all would dilute the log files with unnecessary information, and quickly fill up disk space on a live system. Some actions may be required to be logged for legal auditing reasons but not imply any problems, other actions may be unexpected and result in a warning, and others may signal a critical system failure.

For this reason, most logging packages offer the ability to distinguish between "levels" or "severity" of logs. These levels can then be used when configuring which logs should be tracked, and offer easy filtering of logfiles when diagnosing issues.

For example, the below table is an excerpt from the Python `logging` documentation.

Level	**When it's used**
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

Log destinations

There are a variety of things to do with the logs once your application has created them. This can include:

- Printing to a console window (useful for local development)
- Saving to a file on the server (can be accessed and downloaded for inspection when needed)
- Sending to a specific log-aggregation service (more complex, but can provide running metrics and insights and notifications)

Types of logging

Exceptions

When an exception occurs within an application, it is rarely the case that we want the whole system to halt; usually the exception is caught elsewhere, handled, and the application continues to function. However, this should almost never happen silently - it should be logged to notify developers that an error occurred, and information included to help fix the issue.

A common piece of information included in these log messages (and often included by default in uncaught exceptions) is a "stack-trace", which is the list of nested method calls that lead to the piece of code that raised the exception. It is also often useful to include other data, such as user-ids, or the operation currently being performed.

The whereabouts of this data in your code hierarchy should inform the point at which the exception is caught, so that it can be added to the applicable logs, preventing having to piece together a timeline of events during later analysis of errors. For instance, if a `HtmlConnectionError` exception is thrown in a lower level of code, it can be useful to allow that to bubble up to a middle layer of your application before being caught, to allow more information than just the url to be logged alongside the stack-trace.

Sometimes it is necessary for an exception to be caught, logged, and then re-thrown, to both halt higher levels of the code and allow them to add further information. Care should be taken however to prevent confusion as this can appear like two separate exceptions if not done correctly.

See this article for more information on various exception-logging patterns and their various pitfalls.

Scheduled jobs

Scheduled jobs are particularly important to log as they are capable of failing silently, because there is no request or user interaction to provide feedback to. Some failures might be noticed quickly, for instance in a warehouse system if a job polling for new orders starts failing very quickly your workers have nothing to do. Other failures might only be noticed when they have knock on effects, for instance you might not notice that a job clearing out old data is failing until you run out of disk space or start getting major performance issues. Still other issues may only become clear if there is a security breach, for instance if a job is expiring old credentials.

At a minimum it should be logged when a job has started and succeeded or failed. Many scheduled jobs involve detecting a number of entities that may need updating, and it is often useful to log which entities these are and what ending up being done to them, if anything. These log messages would then be one of the first places to check if issues with background updates are raised.

As above these issues might not be noticeable in a timely manner, so it is critical to proactively check the logs to verify the jobs are running successfully. Doing so manually is generally infeasible so these logs should be monitored as per the next chapter to ensure the jobs are being successfully run.

Performance tracing

Performance issues can make your application unusably slow, expensive to host, or even just stop working as requests start timing out. Fixing these issues is often hard particularly as they might only occur in live usage due to the different load and data sets available in different environments.

Just knowing that your application or even specific requests are slow is not normally enough information to fix them. What is required is an understanding of exactly what step is running slowly. Without this you might spend a vast amount of resources trying to improve performance a little bit everywhere, without knowing that one line of code might be responsible for 90% of the cost.

One way to work this out is to log the time various lines of code take to execute. Attempting to do this everywhere in the code base will result in unmanageably large logs, so it is better to focus on areas you think are reasonably likely to be slow such as making requests to other web services or expensive calculations. Some of this logging can be added by configuring libraries such as HTTP clients to log the time taken every time they are used. If there are performance issues that there isn't enough data on additional logging can be added for future investigation.

Ultimately logging may not be the best way to generate this data. In the next chapter we will see tools that can automatically time the execution of your code, and highlight the most expensive method calls to optimise.

DB queries

For many applications performance issues tend to focus on the database layer. In particular for a large proportion of requests most of the time to fulfil them is querying out data. As the database grows over time this can take longer and longer. Over time this can start causing timeouts, or generally slow performance or high hosting costs.

Often these performance issues will not be apparent in development or testing environments due to the size of the data sets. As such it is critical to get the required information out of the running application, and logging can provide this. Generally the useful information is knowing exactly what step of the processing has slowed down and the exact query that it runs, as this is often not clear from looking at the code.

By configuring the library that is used for connecting to the database it is often possible to globally enable this logging for the whole application rather than needing to add it manually for individual queries.

In the next chapter we cover using an APM (Application Performance Monitoring) service to track down these issues, and then explore how to resolve them.

Managing application logs

Traditionally logs were simply stored on files on the server's hard-drive. As log files became larger and less manageable, log-rotation was developed, where logs are split up, usually into a file per day, with older files gradually getting archived. Whilst this is sufficient for most simple purposes, in a modern distributed architecture with dozens of servers, each occasionally being destroyed and re-provisioned, the collecting, persisting and viewing of log files can get very complex.

A variety of Log Management tools have been developed, such as Loggly and DataDog, which allow easier collation of logs from a variety of sources, along with nicer analysis and visualisations than simply scrolling or using `grep` to search through a text-file.

Logging is also provided by APMs such as Amazon CloudWatch, Azure Application Insights (App Insights) and New Relic, and many logging tools also provide some APM style functionality. Having your logs alongside other application monitoring features makes correlations between metrics and logging clearer, for instance a spate of errors might coincide with a shortage of free memory. It also allows for alerts to be set up based on logged events in the same way as they are for metrics like low disk space.

Open source tools such as the ELK stack (Elasticsearch, Logstash, and Kibana) can be used in place of these Software as a Service solutions, but will require more manual setup than the hosted options.

These all require sending your log events to the service. This might be via a proprietary protocol, by installing the services client library and appropriately configuring your logging library to use it. Alternatively logs can be directed to the logging service at a system level by using syslog. For example loggly provides instructions for integrating with syslog.

Chapter 2

Application Monitoring

While logging is helpful for viewing specific events as they occur in an application, **application monitoring** can provide a much broader view of what is going on in our application. There are two broad approaches to monitoring **manual** and **automated**; both can be useful when trying to understand what is happening in our application.

In this chapter we'll provide a high-level view of some of the techniques that can be used and when they might be appropriate. We'll also mention a few useful tools for automated monitoring.

Manual approaches

There are a number of tools that can be used to debug issues on servers, and to manually verify the health of an application at a point in time.

Profiling

Profiling covers a wide range of different ways of gathering data, however in general it enables basic stats about resource usage over time.

A simple example would be the Windows Task Manager, with this you can see how much memory, CPU disk and network an application is using. With just this simple data, particularly when collected over time, it is possible to identify at the highest level what is causing performance problems. For instance, your app might not be using much CPU, but all of the available memory.

This can help start a search for why your application is having performance issues. Particularly if your application is running in the cloud it might be a better solution to increase the amount of resources available, as this can often be cheaper than the developer time required to optimise the problematic code.

These metrics can also be used for cost control: if your application never uses more than half the RAM or CPU you can probably save on the hosting costs by reducing the performance tier. The inverse can also be useful: during testing you

can run on the cheapest reasonable tier, and only increase the size if the need is demonstrated.

It is also useful to look at profiling while running load tests (as covered in the previous module). If your resource usage remains low during a load test that implies more load can likely to be handled. If it does get worryingly high, or the load test fails profiling can help indicate what the limitation is, so for instance more RAM or CPU quota can be made available. Of course load tests are also likely to affect the data from profiling, so normal usage can't be profiled while a load test is ongoing.

Memory dumps

A memory dump works like a snapshot in time of a running application. These can happen either when an application crashes or on demand, but generating them can cause a running application to hang for a while.

These 'dumps' (as the name suggests) are quite large opaque files that need analysis to be of any use. They are specific to the runtime of the application, for instance a Java Heap Dump is a different format from a .NET Core dump. IDEs will often contain tools for analysing them, but there are also standalone tools depending on the runtime.

These dumps are particularly useful for debugging memory leaks. If your application uses more memory the longer it runs, before perhaps slowing down or crashing with an "out of memory" error, then memory dumps can help track down the problem by giving you a snapshot of all items stored in memory at that particular point in time. This could be by identifying a thread with the most memory usage, or an object that retains a reference to a huge amount of data.

Remote debugging

Many languages let you debug code with your IDE, setting break points, stepping through code line by line and viewing the value of variables. This can also often be achieved remotely by exposing a debug port with a tool like Java's Java Debug Wire Protocol.

This can be tricky to setup in the cloud, as you'll need to tweak firewall rules to enable temporary access to this highly privileged port. When running code in a PaaS, you will also have limited access to expose ports and configure the runtime. How to achieve this varies by cloud, you may need to use specific features of your cloud provider to achieve this, for instance Azure App Services support remote debugging.

This won't be identical to debugging locally. You need to ensure you have the same version of the code checked out as is running remotely. There may be occasions breakpoints are missed due to additional optimisations the compiler performs on release code than local development (debug) code. Performance might also suffer due to network lag and data speeds.

The most important point is that debugging can easily interfere with other users, so it should only be used sparingly *if at all* on production systems. Generally breakpoints stop *all* threads from running, so rather than just stopping the desired request being processed, you may cause the whole application to hang. Even if breakpoints are configured to only stop the current thread others users may hit them. Nobody wants to get back to work on Monday to discover an errant breakpoint has stopped their company taking any orders over the weekend.

Remote debugging can be a very powerful tool, especially if used on a test system. However, because it can be disruptive, using it on a live, production system is **extremely dangerous**. It *might* be appropriate for a tricky issue on a non-critical system, or for an otherwise non-reproducible bug. While you should be aware of this technique, you should only really consider using it in live as a last resort.

Automated application monitoring

The above tools are useful for dealing with issues once they have arisen but are less well suited for continuously gathering information that would help detect or investigate problems that have already occurred.

There are a number of APM (Application Performance Monitoring) services that can continually monitor your application. Perhaps the best known is New Relic. Cloud providers have their own integrated solutions such as Azure Application Insights and AWS CloudWatch.

Uptime monitoring

The simplest form of monitoring a web application is to regularly send requests to it to check that it is running. As this can happen externally from the application, by just configuring a URL (and if required, headers) there are standalone tools to do this such UptimeRobot, but it is also built into many APMs too.

As well as simply alerting you when your site is down they can perform other checks, such as response time, and that your HTTPS certificate isn't about to expire. It is also possible to check specific ports to verify the server is running at all.

Installing an APM agent and setting up configuration

The installation instructions will depend on the APM, and how you are hosting the application. The first step will always be to sign up for the APM, and get a key for it. In the case of App Insights this will be done automatically for you when you create an App Service, and pre-configured as an environment variable. AWS Elastic Beanstalk will automatically install the Amazon CloudWatch log agent. For New Relic there is a shared license key for all your applications.

You'll then need to install the instrumentation. As this integrates with the runtime directly, this is more than just a library that needs to be included with your code. For instance with New Relic there are command line instructions for VMs, and

separate instructions for installing in App Services (https://docs.newrelic.com/docs/agents/net-agent/azure-installation/install-net-agent-azure-web-apps/#site-extention-install) and AWS Elastic Beanstalk (https://docs.newrelic.com/docs/infrastructure/install-infrastructure-agent/config-management-tools/configure-infrastructure-agent-aws-elastic-beanstalk).

On Azure, App Insights will be installed by default for some App Services so this will require no setup, but if you decide to use something else like New Relic you'll need to uninstall it.

Analysing transactions & database queries

One of the most useful features of APMs is the ability to view requests and see which take longest, timeout or error. More generally these are called transactions and can also include scheduled jobs or responding to messages read from queues. This can help identify performance issues, but the real power is the ability to drill into them to see what is taking the time. This is done with a graph that shows each method call and how long it takes, clicking on each line gives the next level down. Ultimately this is likely to have at the bottom a small number of steps that take the vast majority of the time.

Database queries

The APM will (assuming it supports your database access library) show the details of any queries that are executed with performance metrics.

Once the problematic query is known it can be run individually against the database, which should provide tools for working out why the query is slow. Often these tools will suggest a database index, which can easily dramatically improve performance. If there isn't a simple fix by adding an index it'll take some more investigation, but it may be that the query needs tweaking, old data needs archiving or the database needs restructuring. Regardless, detailed information about what exactly is taking the time is critical to focusing on the correct area to fix.

Distributed tracing

Another area that is often slow is calling out to other web services. It is a common pattern in a microservices architecture, or where third party services are used, for calls to one API call to result in downstream calls to another, possibly multiple times.

If the slow service is third party then there isn't much you can do to speed it up directly, other than raise a request. There may however be network issues behind the slowdown that can be fixed, and appropriate caching can speed up repeated requests for the same data.

When dealing with microservices, only having a trace within a single service makes tracking down and resolving issues harder. In order to drill into the downstream application, you'll need to manually locate the request in the other application.

A better idea is to link downstream requests with the one that triggered it. This is called distributed tracing. With this information the APM can let you transparently drill down through the various calls between applications. There are two HTTP request headers (`W3C:traceparent` and `W3C:tracestate`) that enable this to work. These headers are added automatically to supported requests by the instrumentation for your runtime.

Chapter 3

Server Monitoring

When running in a PaaS service the server itself isn't your responsibility. However there are still many times where Virtual Machines are required, and then it is a DevOps responsibility to monitor their health, including if security updates are installed.

In this chapter we'll discuss some of the tools to help you monitor individual servers, networks, and collections of servers.

Built in Linux Tools

/var/log

This is the standard logging location in Linux and contains files for various system wide logging. The various files contain information about different parts of the system for example:

1. `syslog` (or `messages`) as detailed below contains a combination of various other logs
2. `cron` will contain details of any jobs scheduled with cron
3. `auth.log` (or `secure`) contains details of logins
4. `deamon.log` will list running services
5. `httpd` or `apache2` will contain logs from Apache Httpd, if you're using it
6. Various other tools will also log to their own folders in `/var/log` too

Often you'll be accessing these files from the command line on a remote server, so you won't necessarily have tools like a modern text editor to view or search them. Fortunately there are a number of commands that can help handle these logs:

- `cat <filename>` simply outputs the file to the command line
- `tail <filename>` outputs the most recent 10 lines
- `tail <filename> --lines=<number of lines>` lets you see more lines

- `tail <filename> -f` outputs any new lines as they are added, useful for seeing what logging is generated
- `grep` lets you search line by line by piping the output of another command to it.
- e.g. `cat <filename> | grep timeout` to search for the word timeout
- `cat <filename> | grep -C 5 timeout` prints the 5 lines before and after matches, for when some context is useful

  Some of these commands were introduced in earlier modules. They're repeated here as a reminder!

Syslog

`/var/log/syslog` (or for some Linux distributions `/var/log/messages`) is a general log about the system. It can be too verbose to dig in with much detail but does provide a starting point for tools such as `grep`. It can also be used with `tail -f` to see what is logged in real time when performing other actions.

dmesg

The `dmesg` command outputs details about hardware. This can be useful for networking as it will list network adapters and their status. If there are hardware failures then if the hardware is detectable at all attempts to initialise it should be logged here.

Built in Windows Tools

Event Viewer

Windows logs are exposed via the Event Viewer application that comes pre-installed. This contains logs for Windows itself as well as devices drivers, services and applications that integrate with it. Ad-hoc searching and filtering is supported as is creating Custom Views of the logs.

These events integrate with Windows Task Scheduler allowing actions to be triggered when a certain event is raised, such as launching a program or sending an email (although the latter is deprecated, it is possible to achieve by using the "launch a program" option to run a script).

Performance Monitor

Performance Monitor (or perfmon) is a tool for plotting various system metrics (called counters) against each other.

There a number of providers of counters, for instance .NET Framework exposes information about individual applications. However cross platform runtimes (including recent .NET versions) don't integrate with this Windows specific feature.

It's possible to setup alerts and long running reports with Data Collector Sets to see how performance varies over time.

Resource Monitor

Resource Monitor is more orientated around seeing which processes and services are consuming the resources. This can be useful as background processes often have peaks of usage that can affect application performance, and might need reconfiguring to resolve problems. For instance if a scheduled job happens at the same time as a regular virus scan, or if system backups happen during a time of peak load.

Protocol specific logging

Network logging

WireShark is a "network protocol analyser". When running it intercepts traffic giving *very* low level information about network traffic. A common way to use it would be to look at individual packets, of which several make up a request and response, allowing areas such as SSL handshakes to be debugged.

For less low level issues HTTP logging tools such as Fiddler allow the capture of inbound and outbound traffic. With this it is possible to see exactly the requests sent and responses received. It is also possible to edit and replay them if needed to quickly determine issues with requests. Finally responses can be edited for instance if a third party service has a response that your application needs to handle that in practice is hard to force.

Tools like these are *technically* performing a Man in the Middle attack against your own traffic.

Web servers such as Apache and IIS provide logging for inbound requests. In Apache this is generally logged to `/var/log/apache/access.log` (or `/var/log/apache2`) and in IIS is logged to `C:\inetpub\logs\logfiles` but can be configured to be elsewhere. These logs generally contain the IP address requests came from, the URL requested, time taken and the status code returned. It is also possible to configure Apache to add a unique id to this log and setup applications to also log this so it is easy to correlate requests in the access log with messages in the application's log. Similarly other services such as SMTP servers will generate logs of all requests they process.

Monitoring collections of servers

Like with APMs there are also specific server monitoring tools for a centralised view of their status.

The Simple Network Management Protocol (SNMP) is a standard for this monitoring. Windows Management Infrastructure (MI) is built in to Windows, and is an update to what was formerly known as Windows Management Instrumentation (WMI). These can be used in scripts or with reporting tools to see an overview of multiple servers together.

There are commercial cloud hosted tools for this such as New Relic Infrastructure and features in Amazon CloudWatch. There are also open source solutions that need hosting to run such as Nagios Core and Icinga. For these specific software needs to be installed on the servers that are monitored to provide this information.

Chapter 4

Alerting

Monitoring and logging are very useful tools. When an issue is reported they can be vital in identifying the cause, and potentially proving the fix. However, they are limited because by default they are *reactive* systems; you need to actively go and look at them.

Alerting is the practice of automatically and proactively calling attention to potential issues, ideally *before* they become a significant problem for users. This allows you to take act quickly to mitigate, remedy, or even avert issues entirely.

In this chapter we'll explore when you would want to use alerts, and describe some of the common pitfalls with such systems. Finally, we'll discuss what additional monitoring you might want to set up for external systems, or even to monitor the monitoring system itself!

How to setup alert notifications

Alerting policies

The first step is to work out when you want alerts to be triggered. There are a number of different triggers for alerts that can be used:

- if a server cannot be contacted successfully
- breaching a threshold for a metric, e.g. if more than 80% of disk space is used
- searching for log events, possibly also with thresholds, e.g. more than 10 404 Not Found errors in an hour

Each of these has a different use. For instance by alerting when there is less than 20% disk space free it's possible to resolve the issue (by allocating more space, or clearing out old data) before it actually affects any application. On the other hand other metrics like RAM or CPU usage or average response time might indicate ongoing performance issues.

An obvious use for log alerts is for when errors occur. However when setting these up it is worth considering which errors require alerts. The error logs for a web application might include Not Found and Bad Request responses which don't in general represent a problem with the system. A large spike in such errors however might be caused by a bad link or part of the site stopping working.

Some errors also might occur frequently enough that they are handled, for instance adding a retry to an unreliable third party service. In this case individually these errors aren't significant, but an increase in their rate would still be worth investigating.

It is also possible to set alerts for when there is a lower than expected number of messages of a particular type. For instance if a service processing all inbound orders on Amazon records no orders for even a minute that is likely to indicate that something is wrong with either the monitored service itself, or consumers of it. For a lower scale business different thresholds and possibly time periods are required, you might expect an order at least every hour during the day, but getting none between 3 and 4 am is to be expected. If you have scheduled jobs it might also be worth adding monitoring so alerts are sent out if the job starts failing or stops running at all.

Communicating alerts

These alerts can be seen on a dashboard, for instance with App Insights it is possible to assign each possible alert a severity of recent alerts by severity. This might be sufficient for a test environment where downtime isn't critical, and users can all look up the dashboard if something isn't working. For production systems its important that problems are spotted quickly without having to be proactively searched for so you are likely to want to set up notifications.

This is generally done by setting up an alert action and assigning it to the different kinds of alerts. These actions can communicate in a number of different ways, for example:

1. Sending an email to a distribution list,
2. Sending a text to a list of phone numbers
3. Sending a message to a Slack or other chat service group

The option chosen can depend on the preferences of the team, and for text messages in particular if there is an expectation that bugs will be fixed out of normal business hours.

Common pitfalls with alerting

Flooding

If too many alerts are sent that don't need actioning, they will soon be ignored. On the other hand, if something goes wrong and no alert is sent then no one can deal with it. A balance must be struck between sending alerts for potential problems and sending alerts all the time.

It is often a good policy to *initially* send too many alerts, rather than too few. If there are too many false positives - alerts that don't actually require human intervention - then the sensitivity of the alerting system should be reduced until only real, *actionable* alerts are sent.

Examples of false positives could include issues like a spike in `Bad Request` or `Not Found` errors related to an individual user, rather than a systemic problem with the application. It also applies to issues like high CPU load where occasionally short lived peaks are fine, but a prolonged peak, or a series of incidents, are more likely to need fixing.

Often systems become more stable over time after they are first released, especially if they are in maintenance, rather than active development mode. If so it may be advisable to increase the sensitivity of the alerting system again, to avoid missing *real* issues. Broadly, you want to send as many alerts as possible, to ensure issues are detected sooner, without overloading your team.

Where possible alerts should be grouped rather than be sent insistently for a period of time. Sending an email saying a system has gone down at 1:00am, and then another saying it is back up at 4:30am is far better than having dozens of emails, because one has been sent every 5 minutes. An inbox full of duplicate emails is irritating, but worse if a different alert was triggered in that time, as it is then likely that it would be lost amongst the others.

Ownership

Ultimately the purpose of these alerts is for someone to take action. If two or more people try to resolve the problem it is a waste of their time, but also attempts to fix it might interfere with each other. If no one tried to resolve the problem then it will remain broken until support requests start arriving from users. Clearly it is important to co-ordinate who responds to these alerts so exactly one person handles them.

There may be basic tools in the monitoring platform. For instance in App Insights alerts have a state so they can be Acknowledged or Closed. For a more complete system, alerts

could be configured to create issues in a tracking system such as Jira, allowing them to be assigned and go through a fully traceable workflow. Simpler than that, having people reply to an alert in an email of Slack channel saying they'll investigate may be sufficient.

At a higher level, it's also important that having set up alerts, that there are people available to resolve them - and they know that it is their responsibility to do so.

Time sensitivity

Each system will have different required response times, and this needs to be factored in to how alerts are sent.

A video streaming service will have systems to playback video for which any downtime will affect a large number of customers. They will have systems to sign up which will affect a smaller number of customers if it goes down, but also directly affect revenue. At the backend there will be systems to upload new content into the service, this might only be used during the working day, but failures will eventually be noticed by customers expecting new episodes of their favourite series. Finally systems relating to reporting financial data may be able to be down for days without affecting business processes.

If a system requires 24 hour support people will need to be on call to fix issues. Rather than getting them to check emails all night, you are likely to want text messages to go to those on call. An automated escalation process might be required if the main person on call cannot be reached. Alternatively a 24 hour helpdesk might be responsible for getting hold of a on call engineer. This needs to be planned in advance.

For less critical systems, or during office hours email or Slack message may be sufficient.

False Negatives

Another sign of incorrectly configured alerting rules is the **false negative**. A false negative is whenever a production issue requiring our attention occurs, but an alert is **not** sent. This could be because a threshold for triggering an alert is too high, or the underlying issue is a newly introduced (or newly detected) problem.

When bugs are raised by users rather than being proactively highlighted by alerts, it is worth reviewing why no alert was sent. If appropriate, then part of fixing the bug should include setting up alerts for to ensure you are made aware of any recurrence.

Monitoring external systems

Monitoring external APIs

A large set of problems that occur in production systems are not with the system itself but with its dependencies. External APIs go down in much the same way as internal ones do and whilst it might not be possible to directly fix the issues, being able to promptly escalate them to the provider means they will be resolved faster. There also may be mitigation steps to be taken, for instance if a regular job hits the broken API then pausing it until it is fixed may prevent a manual clear up from being applied.

Knowing about third party outages also enables better communication with users, letting them know that a part of the system is down, and can speed up triage of issues as they may not need to be raised with internal support teams if it is clear the origin is external.

One way to monitor external APIs is regularly verifying that a GET endpoint can be successfully hit. If the API only accepts requests from certain IPs or via a VPN then your monitoring needs to attempt the connection over the same network, otherwise it may never be able to successfully connect, or might continue being able to connect when the system itself cannot.

Instead of, or as well as, directly doing this from your application, the check can also be built into a status endpoint on your application. This might only return that your system is fully up if it is running **and** can connect to all external dependencies including database connections and external APIs. If anything goes down or becomes uncontactable then your monitoring will pick up that your application is unavailable and the response should explain which system is down. This has the advantage of checking that the actual networking from the application works, along with with any credentials it may have, but could report the external API is down due to an error in the application itself.

Monitoring monitoring

Your monitoring system can itself go down. If it does, none of your alert notifications will be sent and you might not know.

In the case of a cloud monitoring system, you aren't responsible for fixing it, but that doesn't mean you don't need to know. Fortunately the cloud providers have separate systems for communicating their status. As well as dashboards like AWS Service Health Dashboard and Azure Status it is also possible to sign up for notifications.

When running your own monitoring you may wish to set up a separate monitoring service to check the main one is up. The main

monitoring service can then also check the secondary one is up, and as long as they both don't go down at approximately the same time you can be convinced they are running.

Another issue that can affect your monitoring is if a system stops reporting data to the monitoring service. This will stop you receiving error alerts. If your application should always be regularly logging, for instance if it polls anything, then you can set up alerts to ensure these are received. If not, but your application does run constantly, you could artificially send a heartbeat message and alert if none of these are received. This can also help with verifying services that don't expose an API are still running.

Additional Material

Visualisations/Dashboards

Why do we have dashboards?

Not all stakeholders with an interest in the stability of a system would be interested in getting alerts. Managers generally don't need to see low level email alerts, but on being made aware of an issue would like to check in on it. Prior to meetings they might also want to be able to see an overview of the health of systems. A user of a system might also wish to see the status of it, for example if it appears slow or broken, to see if it is an issue they need to raise or whether the support team are already aware and investigating.

In these cases the users are likely to be less technical, and therefore want a different level of detail than engineers. An example might be showing a red/amber/green status for various components or metrics to see at a glance if a system is healthy or not.

Commercial monitoring software generally has good dashboard functionality as that is the most visible way of demonstrating their value to the people responsible for purchasing decisions.

Building useful views of your data

Dashboards should show data at a level relevant to their intended audience, which could mean a number of different boards for different people. For instance, a team manager might want to see the status of all the systems that their team is responsible for, whereas people higher up and outside of IT are more likely to be interested in whole business areas.

The level of detail should also be dependent on the audience too. Exception messages are not likely to be of interest outside of a team, but the number of errors and trends might be. Similarly, average response times are interesting to people outside IT, but CPU percentage will not be.

These dashboards can also provide some non-technical metrics gathered from systems. For instance it might be possible to display the number of orders raised based on log data. This can also be useful in spotting trends and interpreting the importance of events. For instance if an application starts performing very poorly, but the stats show this was an exceptionally high level of load this can effect how urgent fixing the problem is. Seeing a large spike in use without performance issues can also improve confidence in the system.

Synthetic Monitoring

Beyond basic monitoring to check an application is responding to requests, a more thorough test would be to simulate actual use of an application. This can generate comparable performance statistics as well as more comprehensively verifying the system is currently functional.

For APIs this might be a simple set of (possibly interdependent) requests to simulate an actual workflow. An example of this approach is App Insights Multi-step web tests, and Amazon CloudWatch's API Canary

For websites and applications simulating an actual user journey is more appropriate. In Module 3 we covered using Selenium for automating UI tests, as part of Continuous Integration, but these can also be run against live websites. New Relic supports running these tests as part of your monitoring.

For any such tests it is important that they are setup to avoid unwanted side effects. It might be necessary to develop test accounts to not trigger real world actions such as taking payment or delivering physical products.

ELK Stack / Elastic Stack

One very common logging technology stack is **Elastic Stack**, previously known as **ELK Stack**. We mentioned this earlier in the reading material. The ELK name is an acronym for the stack's components: **Elasticsearch** (a full-text search search engine based on Lucene), **Logstash** (a server-side data processing pipeline), and **Kibana** (a data visualisation dashboard).

Spending some time getting to know these tools may be helpful. There is a useful getting started guide here (elastic.co/start).