logdna

LOGDNA EBOOK

Guide to Using LogDNA Across the Software Development Lifecycle

The eBook will cover how to use LogDNA to debug in development, test during QA and staging, and troubleshoot in production.

INTRODUCTION

Traditionally, logging was most commonly associated with the post-deployment part of the software development lifecycle, or SDLC. Logs typically served first and foremost to help IT engineers find and troubleshoot problems that arose in production.

Today, however, logging can help teams optimize much more than just production-environment application management. And indeed, logging needs to be leveraged across all stages of the SDLC in order to ensure the reliable, continuous delivery of software. Developers, testing teams, and anyone else involved in software delivery must make use of logs and log analysis as one way to ensure the smooth flow of code across the entire SDLC.

With that reality in mind, we've prepared this guide to showcase practical approaches to log analytics at different stages of the SDLC. In the following chapters, you'll find an explanation of why logging across the SDLC is essential in modern software delivery chains, as well as real-world examples of how teams can use LogDNA to streamline three distinct stages in the SDLC: Development, QA and staging, and production troubleshooting.

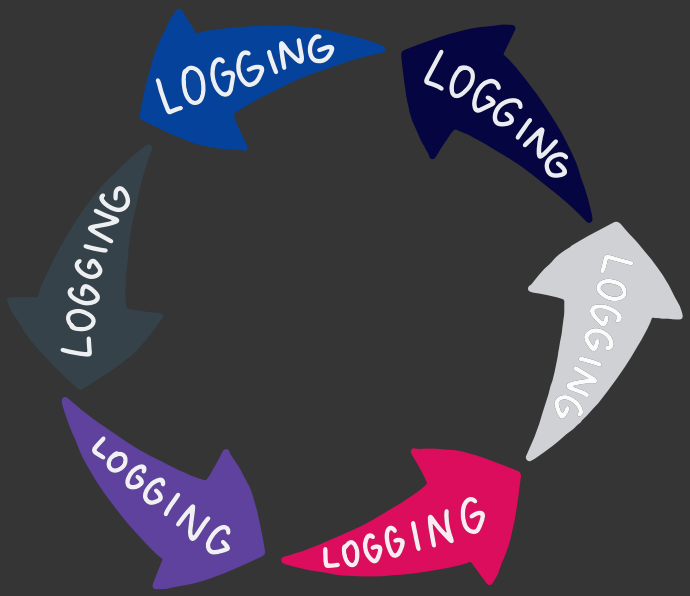


TABLE OF CONTENTS

THE IMPORTANCE OF LOGGING ACROSS THE SDLC

There are multiple phases in the software development process that need to be completed before the software can be released into production. Those phases, which are typically iterative, are part of what we call the software development life cycle, or SDLC. During this cycle, developers and software analysts also aim to satisfy nonfunctional requirements like reliability, maintainability, and performance.

One of the most critical services that developers can include in their applications is logging. Logging is a way to expose contextual information along with the main application runtime. When developers distribute those applications in a live environment, they will collect and store logs, either locally or in an external service.

There's usually not much debate about whether or not to include logging, because almost everyone expects it to be included by default. The discussion that really matters revolves around how to include logging in a way that maximizes its value not only in delivering the platform successfully but also in maintaining it throughout its lifetime.

Keep reading for an explanation of some of the most compelling reasons why logging matters throughout the SDLC.

Logging Improves Software Maintainability

Developers use logs to perform several crucial tasks like debugging, load testing, and performance testing. They almost always capture log errors and fatal exceptions (because those are usually the most important) and then export them to external services like LogDNA for further processing.

However, if you only log errors or fatal exceptions, you can miss significant information. To enable more extensive logging, you can utilize code libraries, which offer a proper set of methods and filters to configure the log information based on a level of sensitivity. Developers can then preconfigure apps to collect or store logs with a specific format and log level.

Utilizing different log levels in different places throughout the codebase allows developers to handle two important things. First, it allows them to configure the sensitivity of the information that gets stored to match the parameters of what they aim to capture. For example, they can include stack traces or detailed context using debug or trace log levels. Second, it allows developers to configure the sensitivity of the log collectors or external services to respond to events that satisfy specific criteria and then forward them into relevant channels.

The aim is to make sure that there will be instances in the code where logging operations are not just distinct actions, but also ways to communicate intent throughout the SDLC. That way, future maintainers can refactor the code easier without breaking any assumptions about the logging operations. Let's take a look at an example of feature-driven logging in Rails in which we try to log user activity information per request. To use the LogDNA Logger instance directly, you would type:

```
logger.debug("SSL Purchase Created for
'#{domain.name}'")
```

Instead of doing that, you need to use a user activity logging module to wrap those events into methods, like this:

```
module ActivityLogging

  extend ActiveSupport::Concern

  def log_ssl_purchase(domain)

     logger.debug("SSL Purchase Created
     for '#{domain.name}'") end

  end

end
```

That way, if you try to change the log details or the level for the SSL Purchase event, you will only have to revise it in one place. Using logging facilities like this is a good way to logically scope them and configure them in a more granular way, thus making the application easier to maintain.

Logging Helps with Migration Phases

You can use logging to record how and when certain method calls are made and also to monitor parts of the code from which you want to migrate. You'll want to log these often and ascertain how much of the existing system is dependent on them. Let's walk through some typical use cases:

Logging Deprecated Features

Our first example is deprecation log messages, wherein we log the usage of functions that are deprecated and slated to be removed in future versions. One way to do this is to wrap the usage of deprecated functions with function decorators. When an attempt is made to call this function, we can log the call and record it into the stream of events. This is what a rough implementation looks like in Javascript using the class decorator syntax for conciseness:

```
@deprecated({useInstead:
'UserManagementService', since:
'v1.0', onCall: (params) =>
logDeprecatedEvent(params)})

class UserApiService {

    static getUsers() { }

}

function logDeprecatedEvent(params):
void {

   Logger.getInstance().info("Deprecated
   call of function with params: " +
   params.toString())

}
```

As far as the client knows, the **`UserApiService`** will still work as expected, but it will now log the call in **`logDeprecatedEvent`** to capture a depreciation event.

Logging Feature Flag Usage

You can log information whenever a feature flag is modified and used to expose new features to users. By doing so, you can gain a more thorough understanding of how the usage of new features affects the existing application performance or reliability. For example, once you've enabled a flag, you might log this information:

```
logger.info("Feature Flag", {"feature_
flag_update": {"name": '#{flag.name}',
"from": "'#{flag.isEnabled}'",

"to":"'#{!flag.isEnabled}'"}})
```

Then, using this as a guidepost, you can record any performance or related changes in the application and measure their success or failure.

Logging Helps with Managing New Work vs Technical Debt

Technical debt is a term that developers use to describe unfinished work that they have postponed due to time or technical limitations. It shouldn't be a frightening word, and it does not indicate poor work ethic. It's relatively common to produce working code with few extra dependencies, quality issues, coupling, or inflexible patterns. Trying to refactor those pieces into a more reusable component may prove to be counter-productive, since it might not be very valuable to the end users, and most of the time, it may not be required.

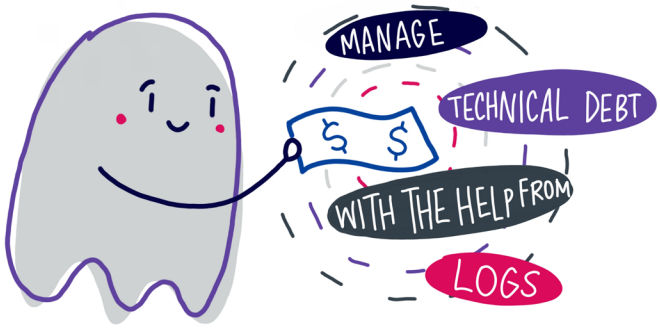
It's at this point in the SDLC that developers might choose to delay any such changes to the existing codebase, thereby managing the technical debt. This does not mean that the code will not change in the future; if it did, then the developers would have to repay this technical debt by spending more time refactoring.

Ultimately, technical debt can become an issue if left unchecked. For example, it can become a problem if you write code without considering reusability, or if you add feature flags here and there without removing them in later stages.

Logs can help manage technical debt by enabling you to compare existing nonfunctional requirements and metrics like performance, error recovery, and availability metrics. For example, developers might touch a piece of code that already exists by including extra features and releasing it to production. If they had log monitors for nonfunctional requirements in place, they could compare the new behavior to the original and assess changes.
For example, did the new code trigger more error messages? Did it reduce build time? Did it reduce memory consumption, or did it increase it? How much time did it take to fix a particular error? All of these are important factors to consider when maintaining new and existing features with logging.

Next Steps with Logging

The aforementioned reasons for including logging are excellent illustrations of just how critical it is to deliver maintainable and reliable software. To realize the enormous potential of logging and improve the SDLC process, you can always rely on a dedicated logging platform that offers a variety of related services – like LogDNA. For example, you can receive alerts and notifications when preconfigured conditions are met using LogDNA Alerts. Their Boards and Graphs give you a completely customizable visualization dashboard that will allow you to present high-level, comprehensive metrics to stakeholders. By pairing those with Time-shifted Graphs, you can see how an app performs across different versions. Start a free trial to see for yourself, or keep reading to learn more use cases for logging.



DEBUGGING IN DEVELOPMENT WITH LOGDNA

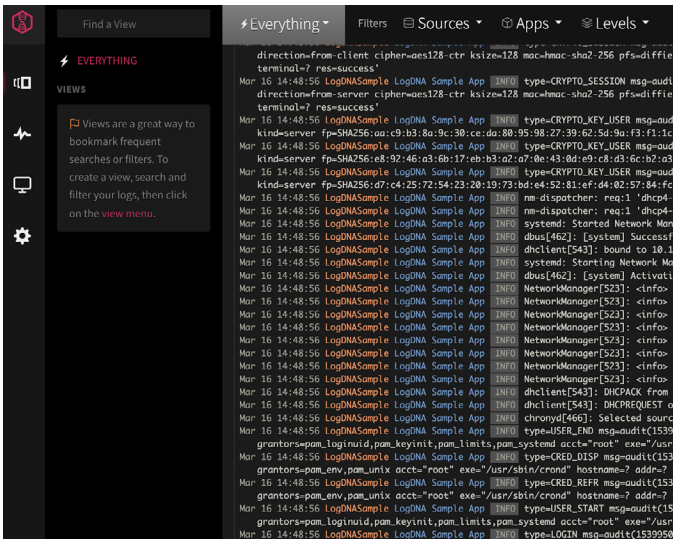
Developing scalable and reliable applications is a serious business. It requires precision, accuracy, effective teamwork, and convenient tooling. During the software construction phase, developers employ numerous techniques to debug and resolve issues within their programs. One of these techniques is to leverage monitoring and logging libraries to discover how the application behaves in edge cases or under load.

Centralized logging gives users access to the information that they need to effectively debug during the development process and LogDNA makes it easy to retain subsets of logs to meet different teams needs. For instance, developers often need access to a true depth of information from their logs, while SREs may be more interested in lightweight logging levels like info and trace. Read on to learn how the LogDNA platform empowers users at all levels of the development process.

Initial Development Environment Setup

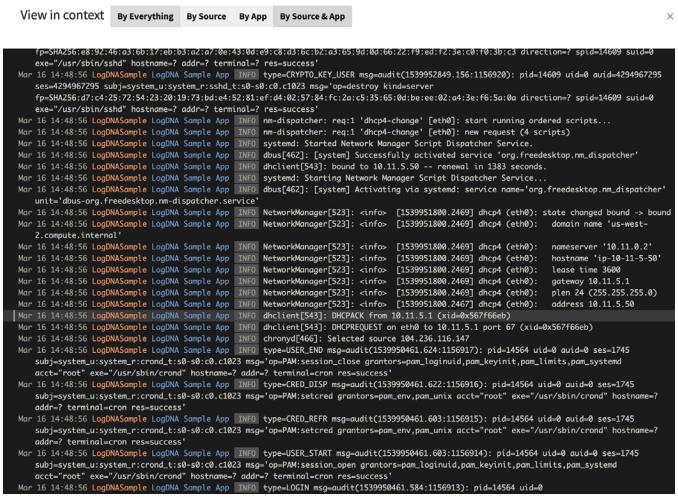
The first thing you need to do is sign up with LogDNA. The process is very smooth. From here, you can explore their dashboard.

On the dashboard page, you have the option to pre-load sample log data or configure an agent collector yourself (or you can do both). If you select the sample data, you can add applications later. Here is what the screen looks like when the sample data is loaded:

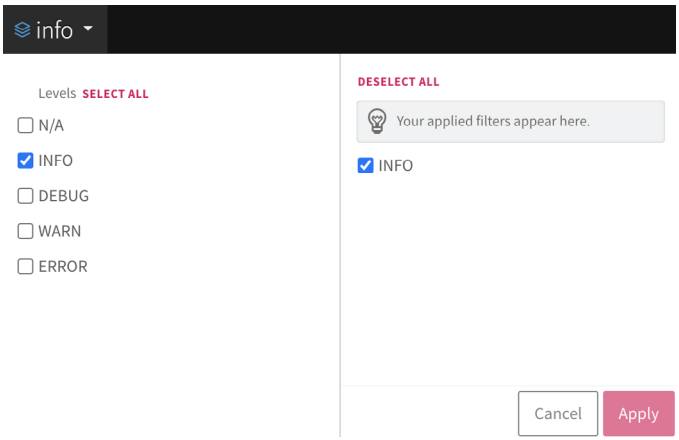


All logs are clearly visible and itemized. When you select a log line, you can view all of the meta field information that was logged at that time. This is due to the automatic parsing of log lines as they are ingested into the LogDNA platform:

You also have the option to view in context. When you click this option, you can see a slice of the logs within the particular context of source, per app, or both. View in Context allows you to see the log lines that have lead up to this event as well as the lines that occured after the event:



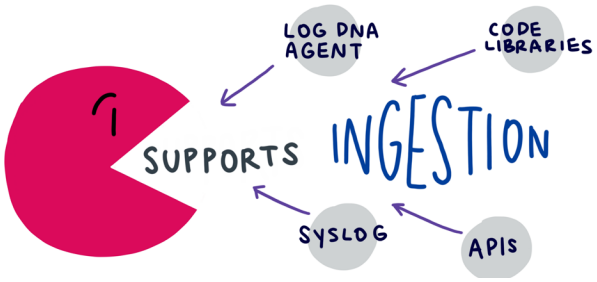
You can also filter the logs by level. This is especially useful for eliminating most of the irrelevant noise when debugging. You can select the filter levels from the dropdown options at the top and apply them to the main view:



Next, we'll show you how to enroll a new application in the platform to test in development.

Enrolling a New Application

LogDNA supports ingestion from multiple sources using the LogDNA Agent, Syslog, Code Libraries, and APIs. In this example, we will enroll a Node.js application sourced from this repo.



You can follow the installation process as explained in the Readme. Then, you will need to hook the LogDNA logger into the Winston.js instance config.

```
$ npm install ip morgan logdna-winston @
types/ip --save
```

Then modify the util/logger.ts file to include the LogDNA configuration:

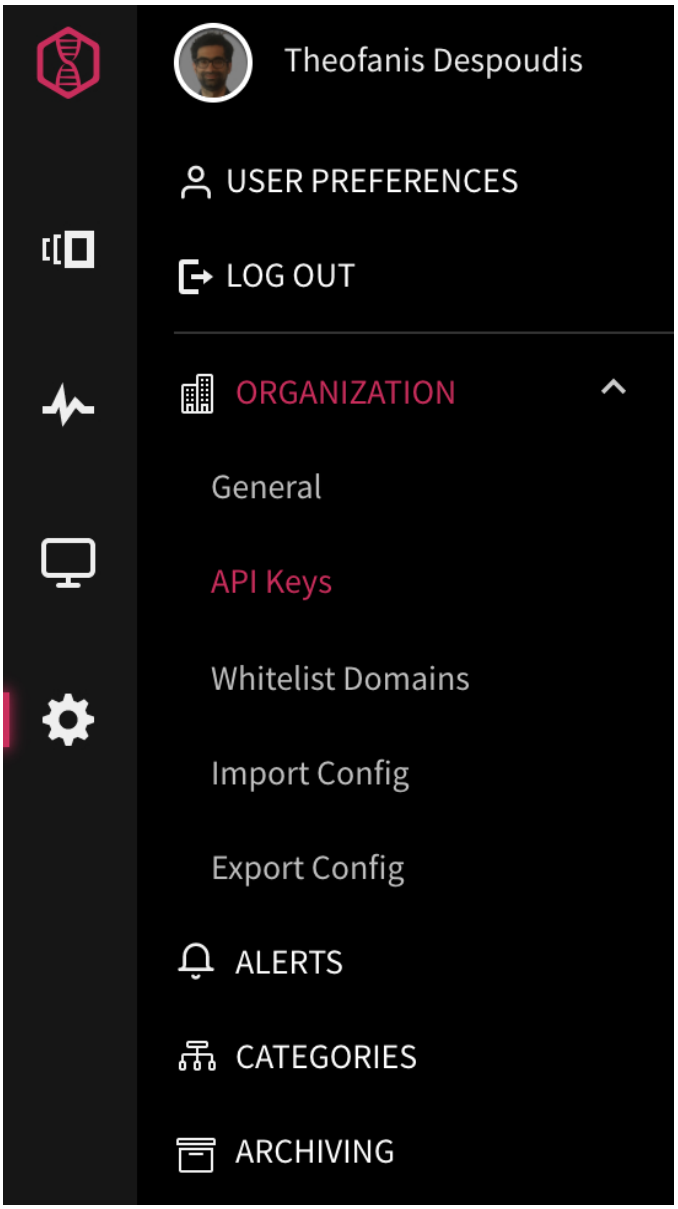
```
import winston from "winston";

import logdnaWinston from "logdna-
winston";

import ip from "ip";


const logDNAOptions = {

key: "b5a09b29ad1d386964c61346108fc981",

     hostname: "localhost",

     ip: ip.address(),

     app: "Typescript-Node",

     env: "Production",

     indexMeta: true

};
```

```
const options: winston.LoggerOptions = {

     transports: [

        new winston.transports.Console({

        level: process.env.NODE_ENV ===
        "production" ? "error" :

"debug"

}),
        new winston.transports.

        File({filename: "debug.log", level:

"debug" })

    ],

};

const logger = winston.
createLogger(options);

options.handleExceptions = true;

logger.add(new
logdnaWinston(logDNAOptions));

try {

     throw new Error("It's a trap.");

} catch (err) {

     logger.error("Log from LogDNA-
     winston", {

          indexMeta: true

          , meta: {

               name: err.name | 'Error'

               , message: err.message

               , stack: err.stack

          }

     });

}

if (process.env.NODE_ENV !==
"production") {
    logger.debug("Logging initialized at
    debug level");

}
export default logger;
```

Then add an empty module definition for the `logdna-winston` package in

```
/src/types/logdna-winston.d.ts

declare module 'logdna-winston';
```

You will need to provide the secret API key for publishing logs in the logDNAOptions. This can be found in the Organization-> API Keys settings:

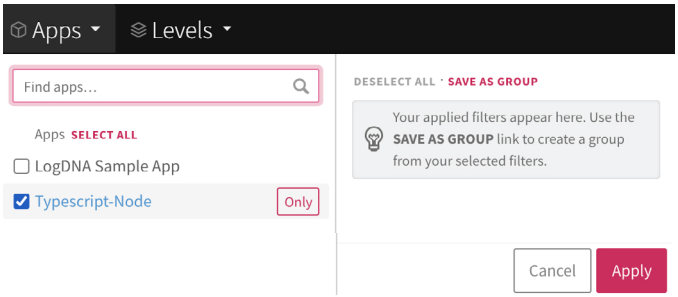


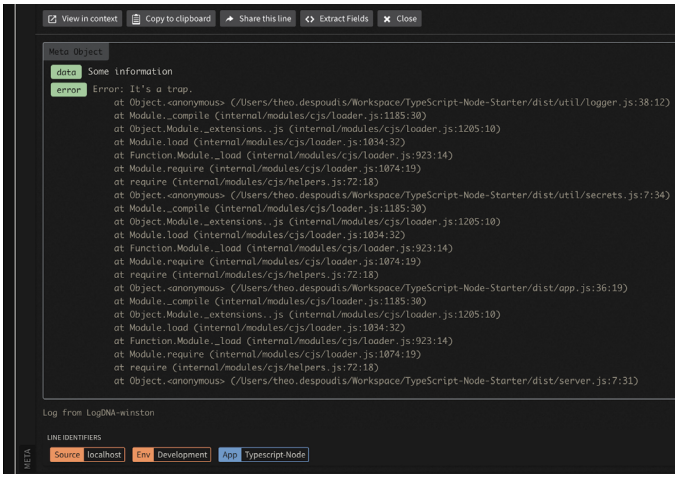
Once you have everything configured, you can start the development server and watch the dashboard as the new logs get populated:

```
$ npm run watch-debug
```

Navigate to localhost:3000 and make sure to enable live monitoring in the LogDNA platform. This can be found at the bottom right of the LogDNA dashboard.

Now you can see the new entries. If you are having trouble finding them, you may want to filter by application first and then select the application name.





Let's take a look at some of the other debugging utilities that LogDNA offers.

Debugging with LogDNA

LogDNA has several options and helpers for debugging applications. Let's explore them briefly one by one.

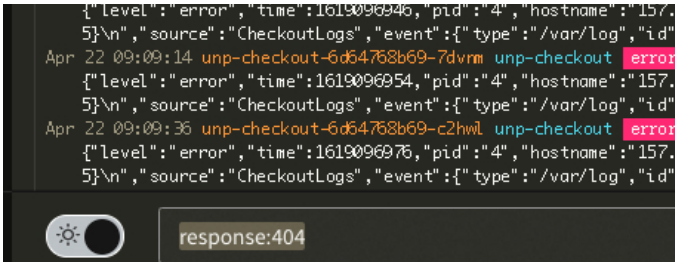
Tracebacks

If you check one of the logs after you have finished the logging configuration, you will be able to see tracebacks. That's because an error was thrown after the logger was configured and propagated into the platform. By lookingat the error trace, you can clearly see that the origin was in the **`/dist/util/logger.js`** file.

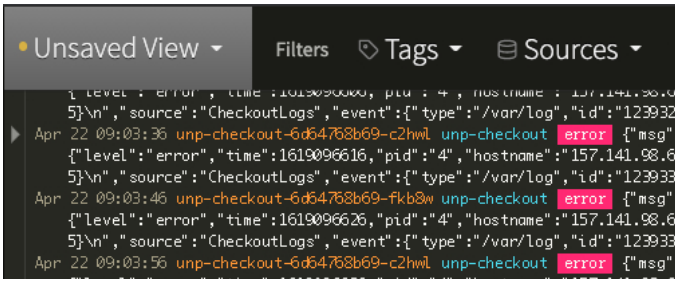
Alerts

Alerts are crucial to any technology as they give us a heads up when something is happening within our environment. With alerts we can get notifications through various means with LogDNA. Out of the box LogDNA supports alerts that can be triggered through email, PagerDuty, and Slack to name a few. LogDNA also supports webhooks for alerting capabilities.

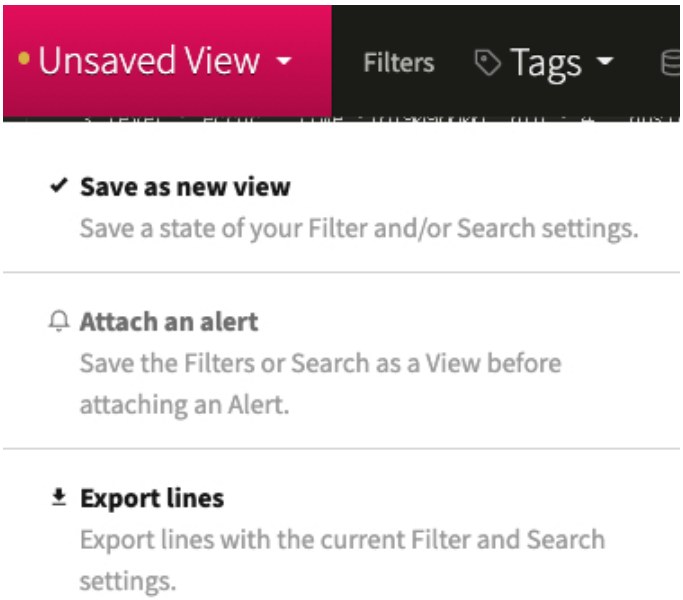
How do we set up an alert in LogDNA? Alerts start when we filter down our logs to a specific query we are interested in. Filtering can take place through several means within the platform, but for this example we will use the natural language query syntax to filter down 400 response errors that are typically specific to web applications.



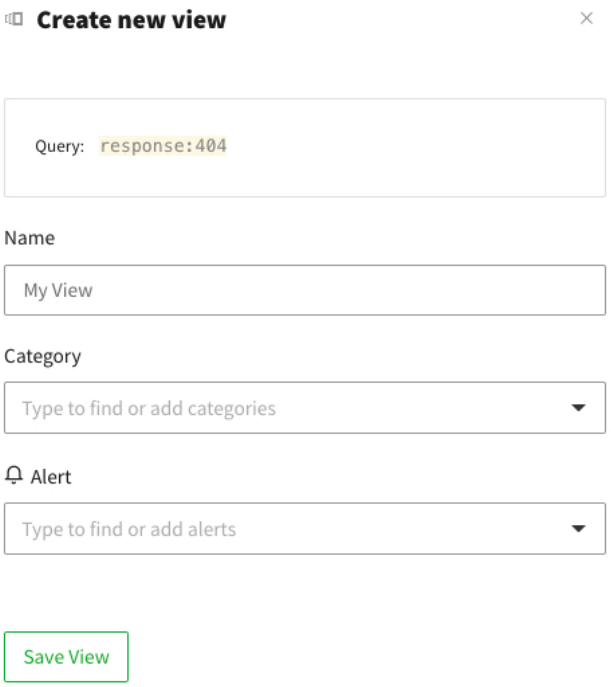
Now that we have our filter in place it's time to set up our alerting. For that you will notice that you will see your View change to 'Unsaved View' at the top of the Views page.



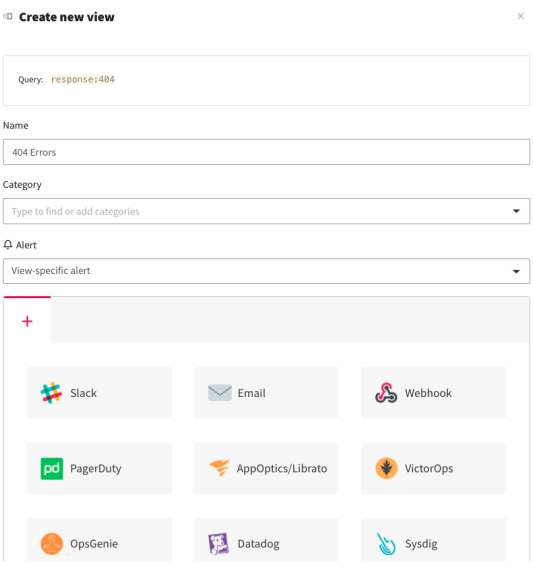
Clicking on the 'Unsaved View' will provide us options to save the View and attach an Alert to it. If you already had a saved View you would have the ability to attach an Alert to that existing View from this menu.



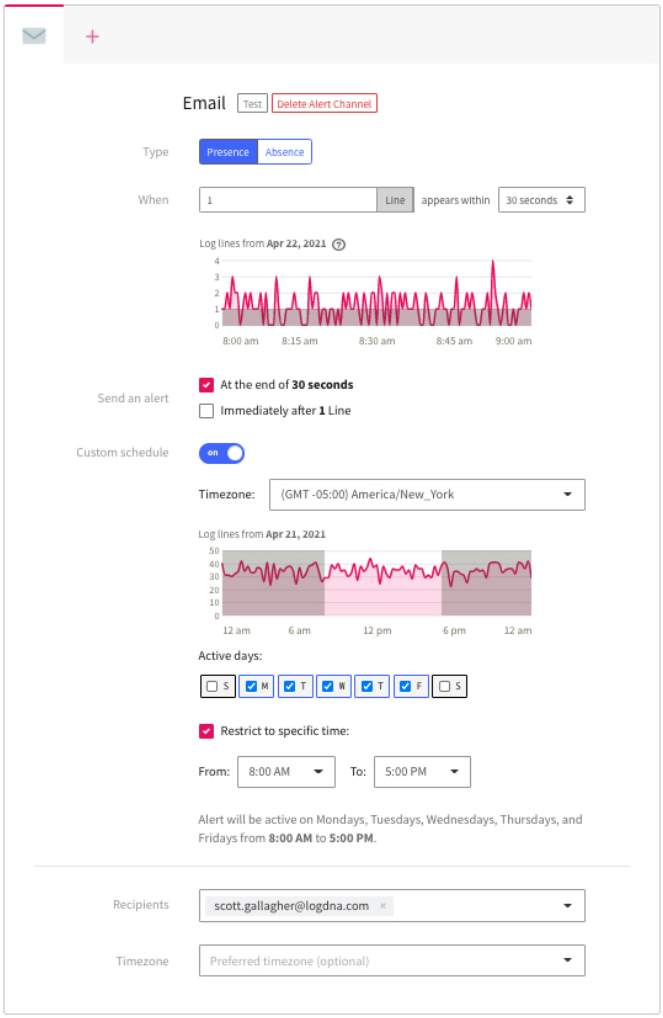
Let's look at what happens when we click on 'Save as new View'.



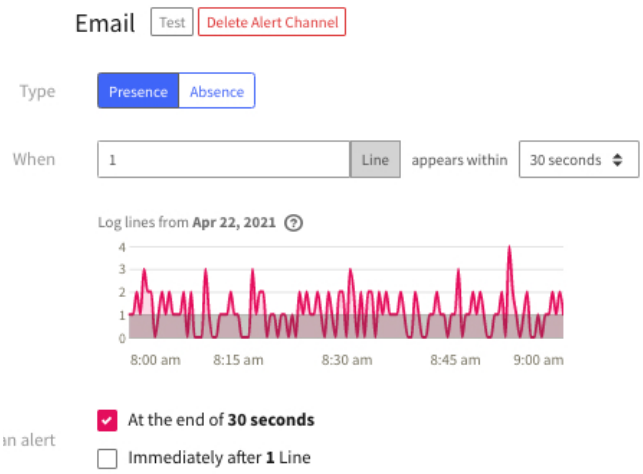
Within the pop up window we can give the View a name, add it to a category, and attach an Alert to it. When we go and attach an Alert to our View we are presented with the screen below.



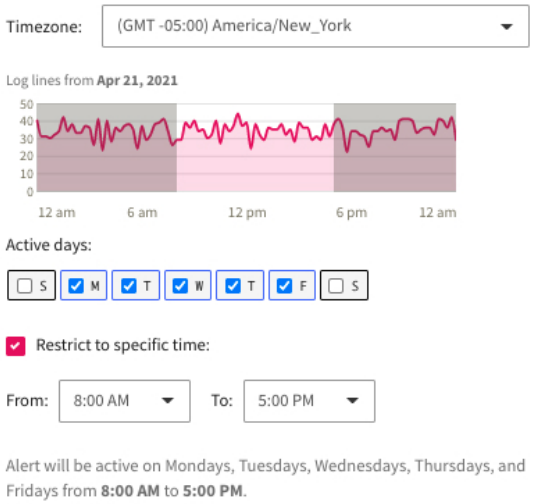
From here we can see the various alerting options that are provided out of the box, and also see the Webhook option as mentioned before. We will select email for our first option.



There is a lot to take in with the above screenshot so let's walk through it piece by piece.



The first section is all about alerting off either the presence or absence of log lines. Think of presence as meaning "when I see a defined number of lines come in during a specified period, I want to be alerted to this." Absence would be the opposite of that. It would mean "I'm expecting my application to generate X number of log lines and if it dips below that, then I want to be alerted as there may be issues with my application continuing to run and accept calls properly." We can also see when this Alert will be triggered based on our input with the gray line that runs across the display.

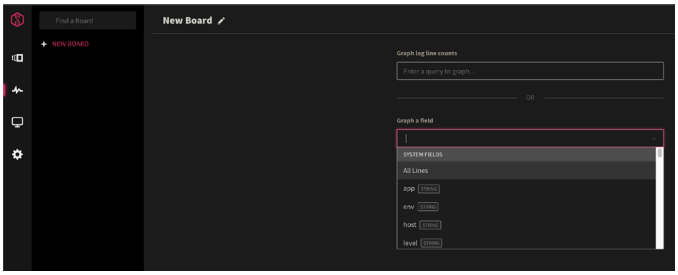


Next we can create custom schedules that define when this Alert is to be triggered. For this example we can specify typical working hours of Monday through Friday from 8:00 am to 5:00 pm. This is useful as we can create alert escalations that are sent to one place during normal operation hours and another place after hours or on the weekends.

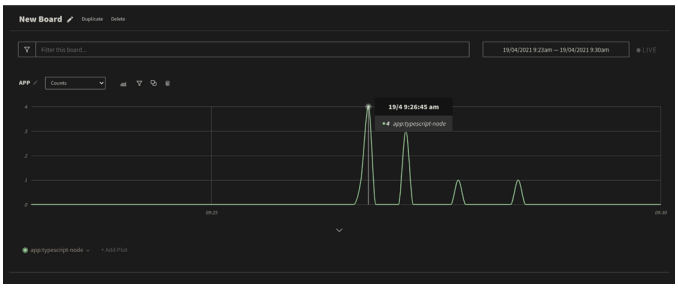
Alerting is crucial these days with such busy schedules, remote working, and it helps avoid things like context switching where you'd have to be monitoring a web UI all the time.

Boards and Screens

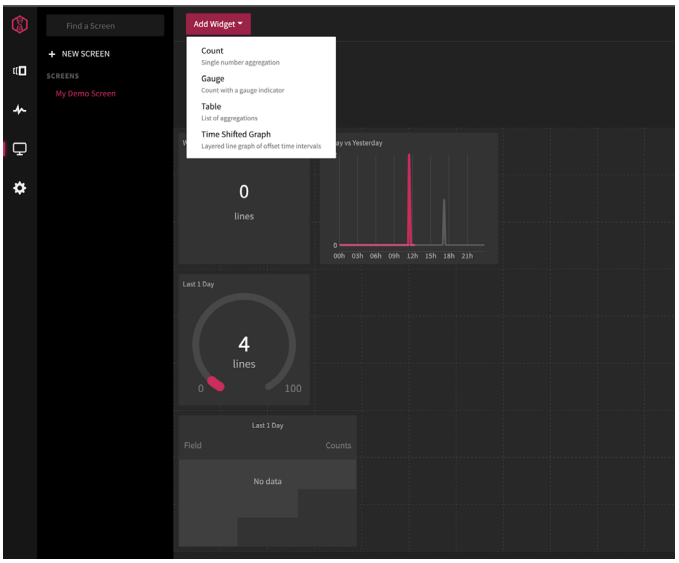
After setting up Alerts, you can create your own Board with custom widgets. For example, you can add a widget that uses only logs from a particular application:



In this example, if you select app and Typescript-Node, you will see the following graph:

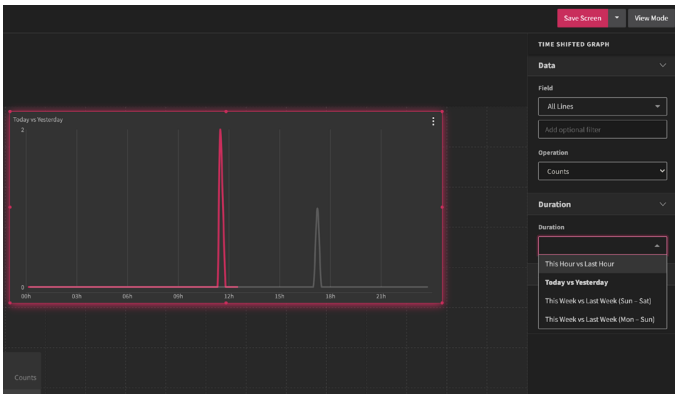


Screens are similar to Boards, but they give you a birds-eye view of your widgets. You can place them wherever you like.



Time-Shifted Graphs

After you've written some application logic, you can revisit the application in specific time intervals to check if the reliability has improved. This can be accomplished by using Time-shifted Graphs. With this feature, you can compare log events across two different time spans. To do so, you begin by selecting a widget from a screen. Then, using the sidebar options, you can change the duration field to provide valuable insights about the rate of events:



These Graphs are excellent for development, since they demonstrate the general tendency of the log events after new test cases have been written or major code changes have been implemented.

Next Steps with LogDNA

This chapter offered a brief tour of the main features of LogDNA's platform that cater to developers. We showed you how to review tracebacks, view in context, use Live Tail, and set up LogDNA Alerts for fundamental errors. Together with Boards, Graphs, and Screens, this platform gives developers a comprehensive set of tools for debugging applications. You can also take it to the next level by using LogDNA for production environments – but we'll explore that topic later in this eBook.

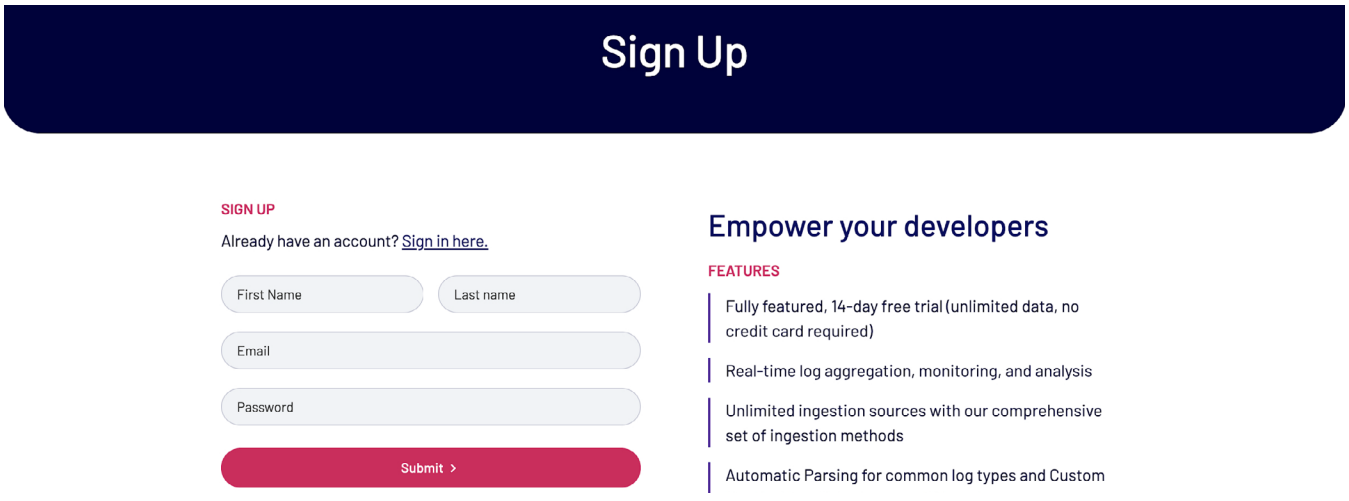
USING LOGDNA FOR QA AND STAGING

The purpose of the QA and staging part of the SDLC is to test software and – assuming it meets quality requirements – prepare it for deployment into production environments. To do this, engineers need to be able to identify performance or reliability issues that exist within the application. At the same time, though, they must ensure that their data is actionable, and that it helps them quickly fix issues, in order to avoid holding up the SDLC.

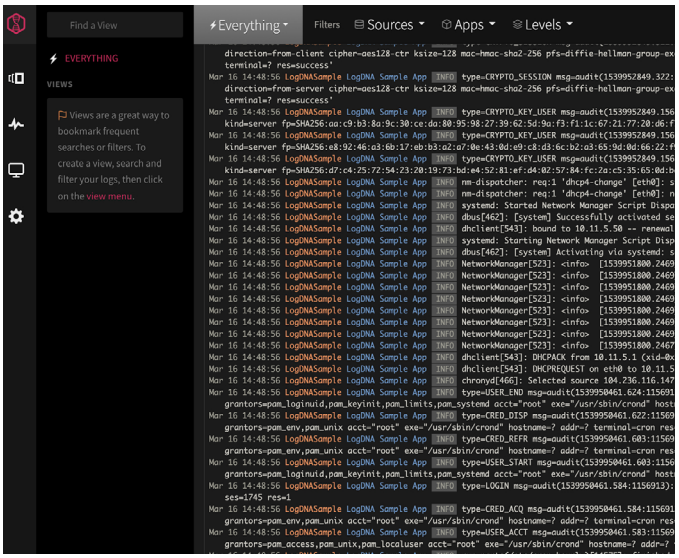
For QA and staging, then, logs must provide refined data that allows engineers to anticipate what will happen in production, and helps them get to the root cause of any problems they detect. This chapter explains how to use LogDNA for this purpose.

Setting Up A Staging Environment

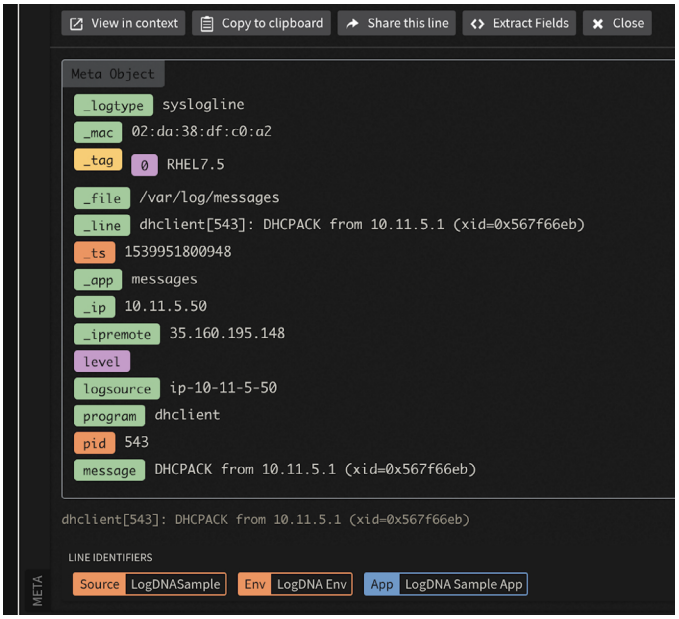
The first thing you need to do is sign up with LogDNA (if you haven't already done so when working through the tutorial in the previous chapter). When this process is finished, you can explore their dashboard page:



On the dashboard page, you have the option to pre-load sample log data, or configure an agent collector yourself (or you can do both). If you select the sample data, you can add applications later. Here is what the screen looks like when the sample data is loaded:



All logs are clearly visible and itemized. When you select a log line, you can view all of the meta field information that was logged at that time:



Next, we'll show you how to enroll a new application in the platform and run it in production mode so that we can perform tests and log events.

Enrolling a New Application

As explained in the previous chapter, LogDNA supports ingestion from multiple sources using the LogDNA Agent, Syslog, Code Libraries, and APIs. In this example, we will log data from a Node.js application sourced from this repo.

(This is the same application that we enrolled in the tutorial in the preceding chapter, but the enrollment process is slightly different in this chapter because here we are setting up a staging environment that is designed to mimic production, rather than a development environment – so, to get the most out of this chapter, it's best to enroll the application again using the steps below, rather than reusing the one from the previous chapter.)

You can follow the installation process as explained in the ReadME. Then, you will need to hook the LogDNA logger into the **Winston.js** instance config.

Install the following packages:

```
$ npm install morgan @types/morgan ip logdna-winston @types/ip --save
```

Then modify `the util/logger.ts` file to include the LogDNA configuration:

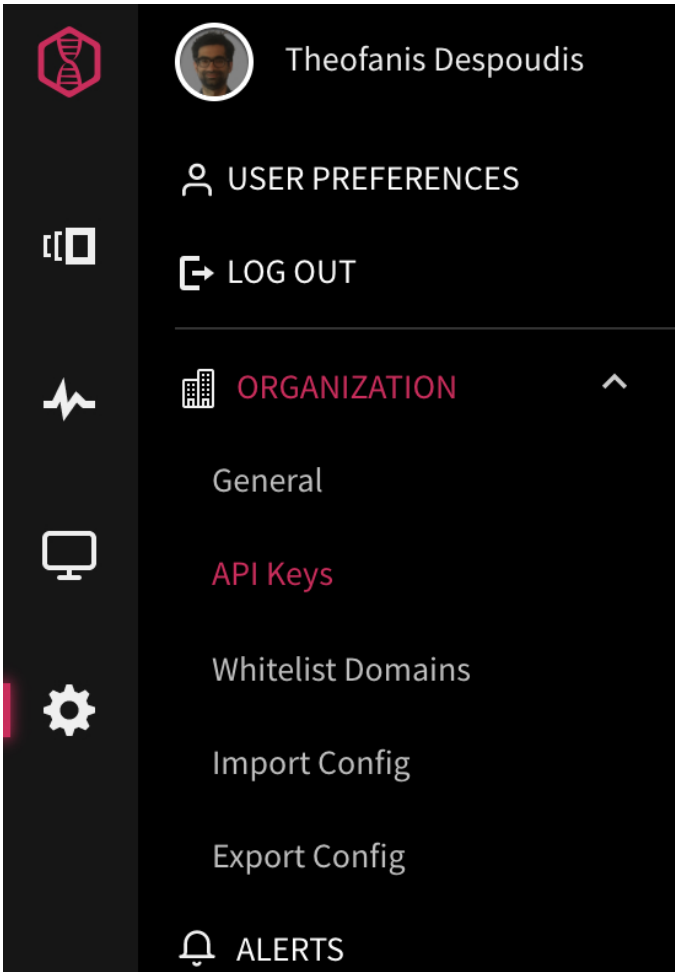
```
import winston from "winston";
import logdnaWinston from "logdna-winston";
import ip from "ip";
const logDNAOptions = {
   key: "LOGDNA_KEY",
   hostname: "localhost",
   ip: ip.address(),
   app: "Typescript-Node",
   env: "Development",
   indexMeta: true
};
const options: winston.LoggerOptions = {
```

```
  transports: [
    new winston.transports.Console({
        level: process.env.NODE_ENV ===
        "production" ? "error" :
"debug"
    }),
    new winston.transports.File({
    filename: "debug.log", level:
"debug" })
  ],
};
const logger = winston.
createLogger(options);
options.handleExceptions = true;
logger.add(new
logdnaWinston(logDNAOptions));
if (process.env.NODE_ENV !==
"production") {
  logger.debug("Logging initialized at
  debug level");
}
export default logger;
```

Then add an empty module definition for the `logdna-winston` package in

```
/src/types/logdna-winston.d.ts

declare module 'logdna-winston';
```

You will need to provide the secret API key for publishing logs in the logDNAOptions. This can be found in the **Organization-> API Keys** settings:



Once you have everything configured, you will need to start the production environment server.

You want to get close to a production instance even when connecting to MongoDB or Social Login; for example, in the demo application there are .env options for **`MONGODB_URI`**, **`FACEBOOK_ID`** and **`FACEBOOK_SECRET`**. Those are used to log in via Facebook and for connecting to a remote MongoDB instance. We recommend a production ready MongoDB server from MongoDB Atlas and using a testing account for the Facebook Login.

You will have to build the assets first and then start the server. This can be done with the following commands:

```
$ npm run build && npm run start
```

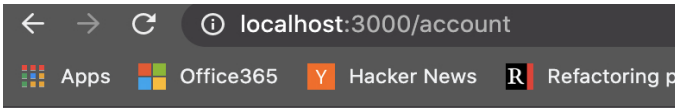
You will see the following events logged into the console:

```
(node:47780) Warning: Accessing non-
existent property 'MongoError' of module
exports inside circular dependency

(Use `node --trace-warnings ...` to show
where the warning was created)

App is running at http://localhost:3000
in production mode
```

You can navigate into http://localhost:3000/ and interact with the page.

Testing the Environment with LogDNA

If you try to sign up a new account with email, you may encounter an internal server error when navigating to the account page:




```
Internal Server Error
```

Currently the logger does not capture any information about the error. Let's use the logger to record those error messages so we can inspect them in the LogDNA dashboard.

For every render method, you need to add an appropriate handler. For example in **`src/controllers/home.ts`** replace the code with:

```
export const index = (req: Request, res:
Response, next: any) => {
   res.render("home", {
      title: "Home"
   }, function(err, html) {
      if(err !== nutll) {
         next(err);
      } else {
         res.send(html);
      }
   });
};
```

With errors captured in the template, we need a middleware function to log errors. You can do that by including the following handler in **`src/server.ts`**:

```
app.use((err: any, req: any, res: any,
next: any) =>
   { if (err) {
      logger.log({
         level: "error",
         message: err.message
      });
   }
   next(err);
});
```

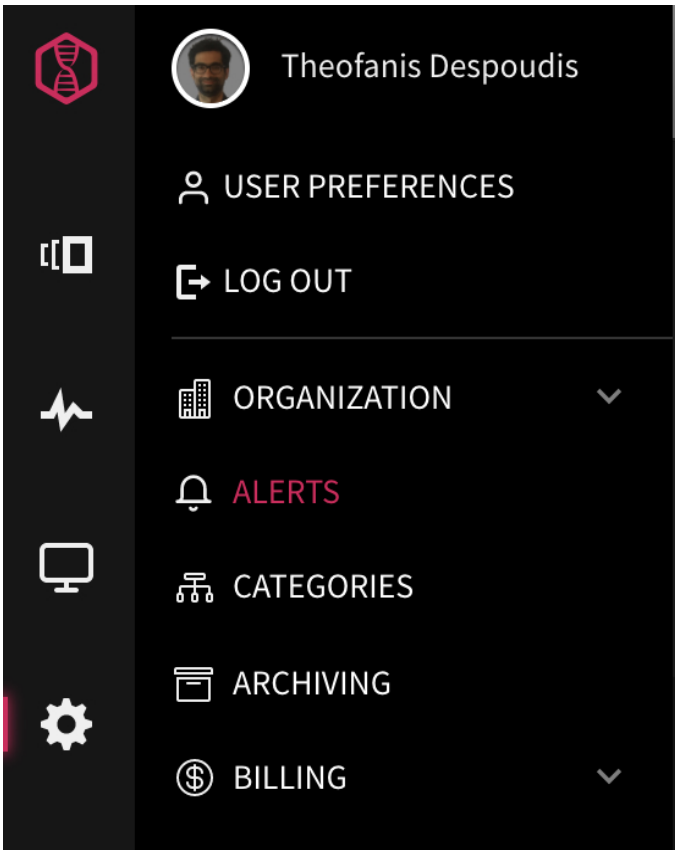
You may also want to capture access logs. You can add a **morgan** logger middleware in **`src/app.ts`**:

```
// eslint-disable-next-line
// @ts-ignore
app.use(morgan("combined", {
   stream: {
      write: (message: string): void => {
         logger.info(message.trim());
      }
   }
}));
```

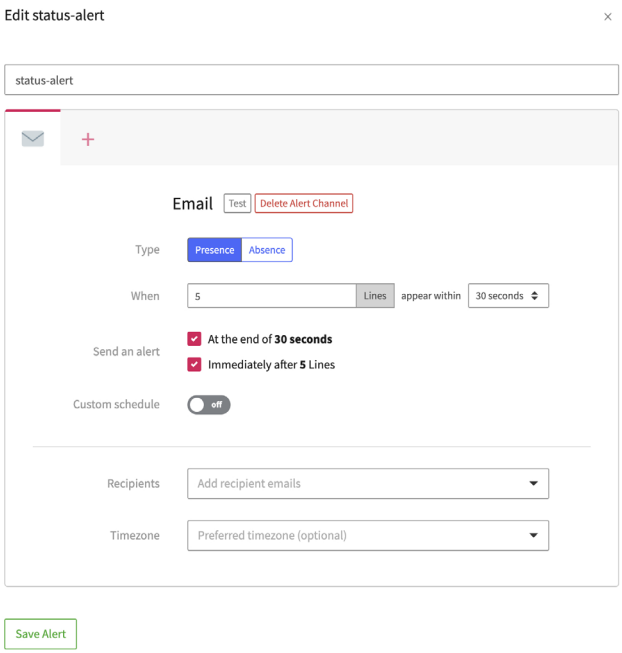
Now you can inspect the logs in the main Live Tail view:

Alerts

You can set up Alerts so that certain log lines trigger notifications. This is done in the Alerts sidebar option:

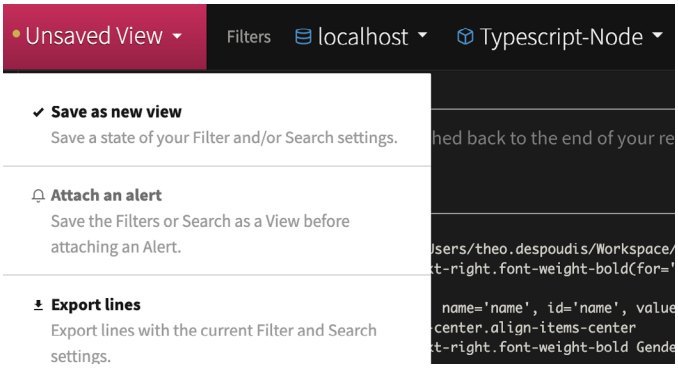


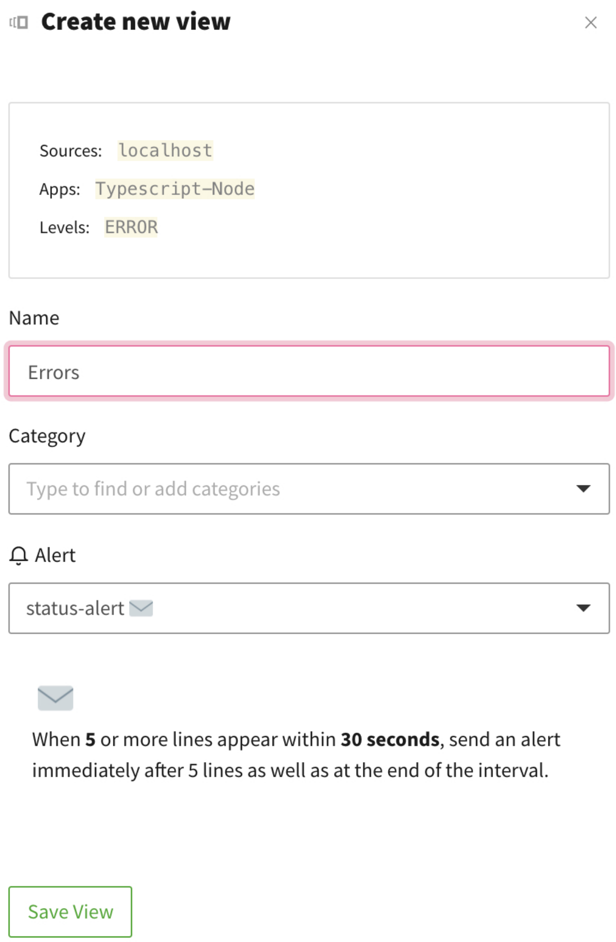
Let's first create an Alert that triggers an email after at least 5 events are logged. When you click 'Add Preset' you will need to fill in those details:



Once created, you want to attach a **View** to it. A **View** is a saved filtering of logs based on some criteria like status codes for 400, 500, or other errors. In this example, we create a View for the template errors we found earlier.

On the Logs View, you want to select the appropriate filters on the top bar. Select host=localhost, Application=Typescript-Node and level=Error and click on the **Unsaved View -> Save as a new view** and select the Alert we created before.

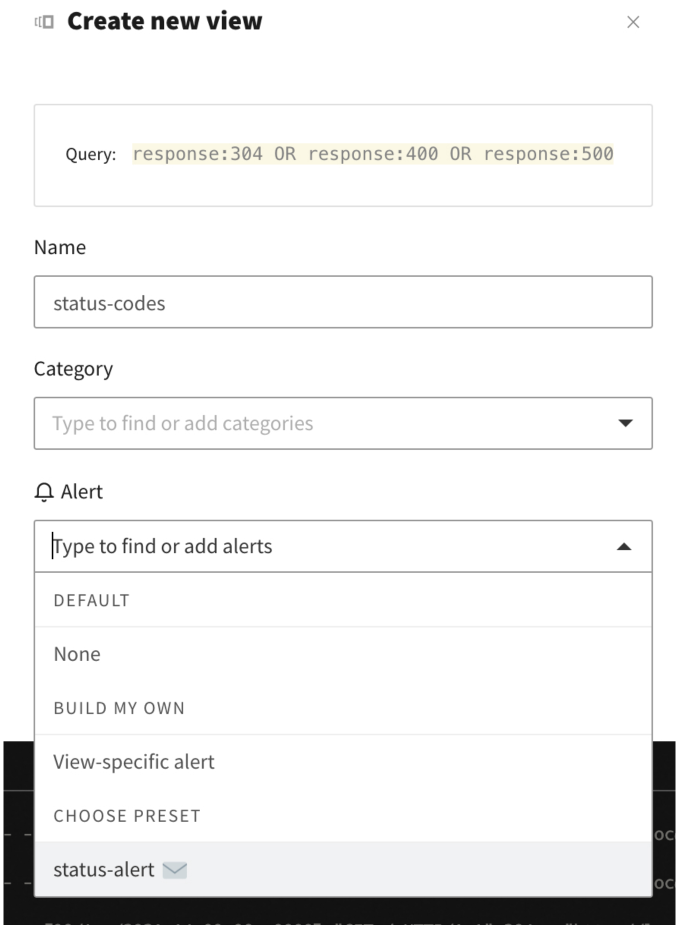




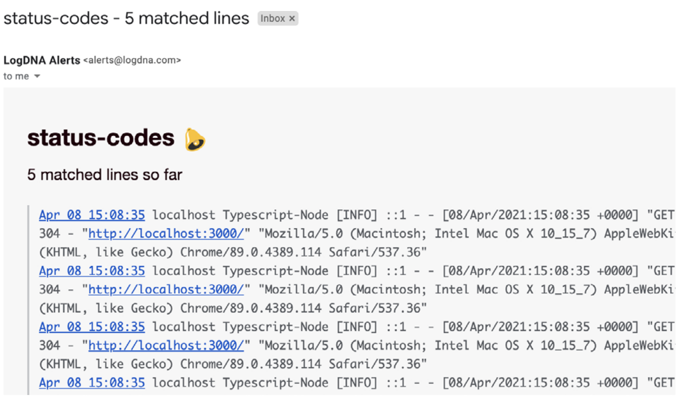
Let's add another one for specific status codes like 304, 400, and 500. You may want to use the bottom search bar. Enter the following query and save it as new View:

```
response:304 OR response:400 OR
response:500
```

The View modal shows the parameters you selected and which Alert to use:

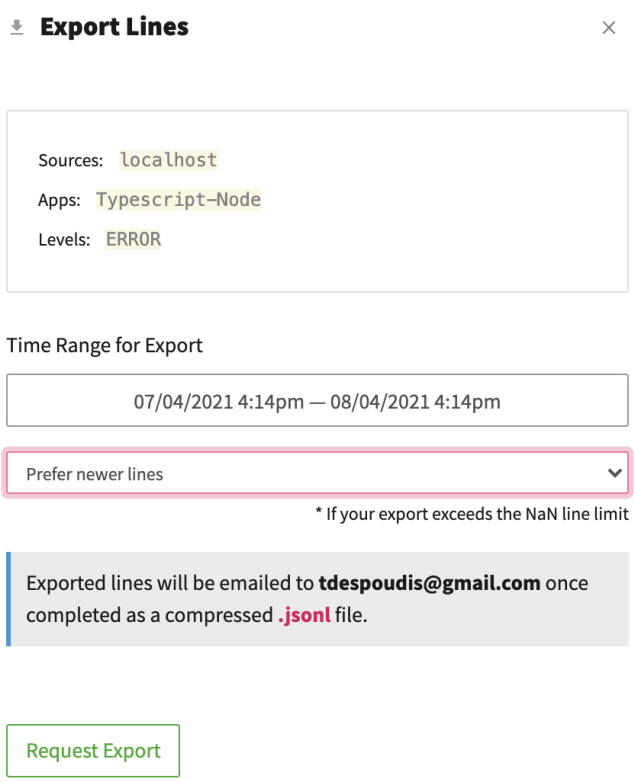


Now you will get email notifications when you exceed those alert thresholds:



Sharing Views with Developers

Now that you've created some custom Views, you can share them with developers or related parties so they can inspect them on their own. You can use the **Export Lines** option to grab an export when selecting an existing View from the top Bar:



You will get an email containing the lines in jsonl (JSON Lines) format. Send those files to the developers; they can inspect them with the following command:

```
> cat
export_2021-04-08-15-16-00-827_544dfc58-
785c-4a1f-9151-8b480dd038ef.j sonl | jq .
```

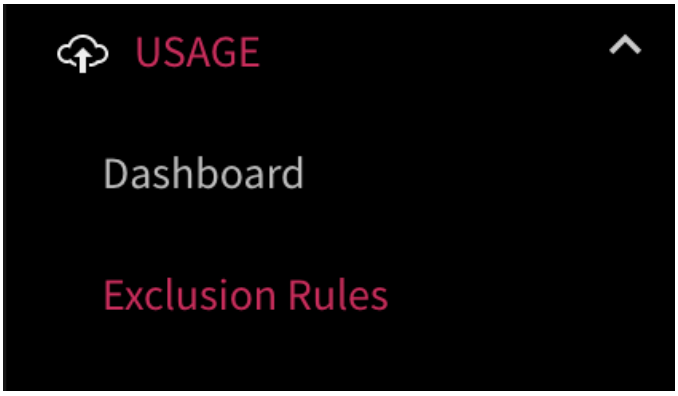
If the developers have access to the Dashboard, you can share the link to the View as well. This is an example link:

https://app.logdna.com/b5a09b29ad/logs/view/d02237cb23

How to Exclude Log Lines Before and After Ingestion

When logging information in requests—especially for environments that mimic the production site—it's significant not to capture any sensitive data such as login credentials or passwords. You can define rules, using regex (regular expressions), to control what log data is collected by the agent and forwarded to LogDNA to prevent those kinds of events from ever appearing in the LogDNA dashboard (check out this GitHub repo to learn how).

You may also want to filter out logs by sources, by app, or by specific queries using an Exclusion Rule. These are great for excluding debug lines, analytics logs, and excessive noise from logs that aren't useful. You can still see these logs in Live Tail and be alerted on them if needed but they won't be stored. To start, find the **Usage-Exclusion Rules** option in the sidebar:

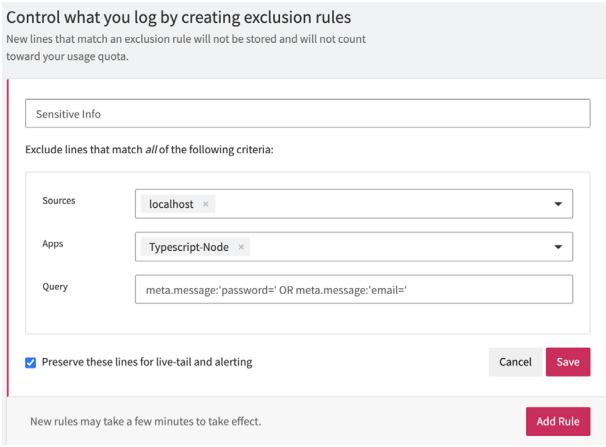


Then you will need to fill in some information about the rule. You may want to find specific log lines that match the query first. Here is an example with a message matching the following lines:

```
meta.message:'password=' OR meta.
message:'email='
```

This tells us to ignore lines with messages containing the strings `password= or email=`. Note that this can still capture those lines, it just doesn't store them.

It's a best practice for users to redact PII and sensitive information from their apps before sending them to LogDNA; so if you want to capture the information but not the sensitive data, you will need to redact those fields from the application as well or mask them at the agent level.



Examining automated tests for failures

You are not limited to using the logger instance for runtime information capture. You can use it for CI/CD pipelines and test cases as well. This would give you a convenient way to capture test results all within the LogDNA dashboard.

Because you may want to capture specific information within a CI/CD pipeline, you want to attach a meta tag or a different level on it. At first, when you run test under CI/CD, you want to set a **`CI=true`** environment variable and pass it into the logger instance config:

```
const isCI = process.env.CI === "true";
const logDNAOptions = {
  key: "b5a09b29ad1d386964c61346108fc981",
  hostname: "localhost",
  ip: ip.address(),
  app: "Typescript-Node",
  env: isCI ? "testing" : "production",
  indexMeta: true
};
```

Then you may want to create a new View for that Environment. In order to capture errors you may want to configure a custom reporter to log any failed test cases. This is beyond the scope of this tutorial, but you can see an example using Jest here:

https://jestjs.io/docs/configuration#reporters-arraymodulename—modulename-options

Once you get those logs captured, you can connect them with Alerts and Boards as well. This will help you visualize these errors and correlate them with recent code changes.

Next Steps

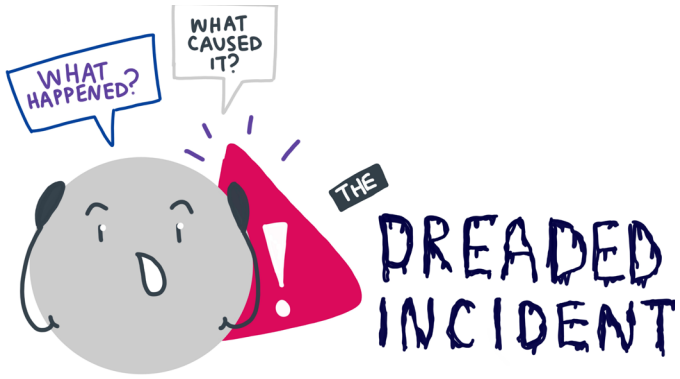
In this chapter, we saw how to leverage the LogDNA platform for QA and staging environments. As a rule of thumb, those environments should match exactly the production versions in both functional and non-functional requirements. Additionally, when running integration and system tests, those test logs should be queryable in case of failures. Using LogDNA Alerts, Boards, Graphs and Screens can help catch and visualize those errors in correlation to any recent code changes. Lastly, using saved Views, you can delegate important information to developers when trying to discover significant problems or performance bottlenecks. Feel free to try the LogDNA platform at your own pace.

USING LOGDNA TO TROUBLESHOOT IN PRODUCTION

In 1946, a moth found its way to a relay of the Mark II computer in the Computation Laboratory where Grace Hopper was employed. Since that time, software engineers and operations specialists have been plagued by "bugs." In the age of DevOps, we can catch many bugs before they escape into a production environment. Still, occasionally they do, and they can spawn all kinds of unexpected problems when they do.

In addition to software bugs, today's modern systems encounter other problems as well. And these problems collectively manifest as issues in production. Or, as some might refer to them, the dreaded incident!

When an incident occurs, it's not always readily apparent why it happened or what caused it in the first place. This chapter explores some of the different situations that can result in a production incident, and we'll investigate how you can uncover these situations using LogDNA. Troubleshooting a production incident shares many similarities with how a medical professional might approach a patient's diagnosis. We look at the symptoms, we run tests, and we reach a diagnosis by drawing on our experience, the experience of others, and sometimes even relying on the process of elimination.



Detection and Recovery

Production incidents have two distinct phases: detection and recovery. Many teams track ***Mean Time To Detection (MTTD)*** and ***Mean Time To Recovery/Resolution (MTTR)*** metrics. MTTD is a measure of how long it takes for a problem or incident to be identified and acknowledged by the responsible team. MTTR then measures the amount of time before the team can identify and rectify the underlying cause.

This chapter is not about tracking metrics; however, it is essential to measure and track their effect on our results when we're investigating and investing in new ideas and tools. Metrics such as MTTD and MTTR provide visibility into how effectively the organization manages the incident process and how valuable different approaches and toolsets are within that process.

The Complexity of Modern Systems

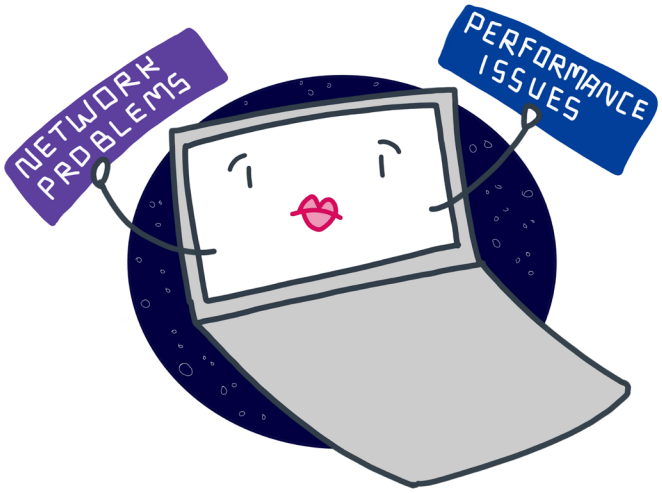
Modern computer systems have begun to rely on microservice architectures, patterns for high-availability, and public cloud offerings. We build highly scalable systems that take advantage of ever-improving connectivity and the availability of third-party infrastructure and services. While the benefits are apparent, we find ourselves managing increasingly complex systems. Let's consider the example of a web application that becomes unresponsive.

Networking Problems

Between the user and our web application, there is a complex web of connections running on private and public infrastructure. Service outages and hardware failures can easily affect a user's ability to connect to our systems. Even within our systems, network calls travel between different instances and services. Our applications and services may be running perfectly, but if the connections between them are broken or experience performance degradation, the system won't work as expected.

Performance Issues

Performance issues can occur due to hardware problems, constraints in frameworks and infrastructure we rely on, or poor design within the code itself. As the amount of traffic our applications handle increases, many of these problems grow exponentially, resulting in performance degradation or connection timeouts.



Why Centralized Log Management is Essential

Logging has always been a central component of understanding and troubleshooting applications. System and application logs provide an essential window into how data is processed and transferred, as well as the infrastructure's performance. Distributed systems complicate log management because there are so many different places where logs are created and stored.

Centralized log management solutions, like LogDNA, give you the ability to collect and aggregate all of your logs in a central location. However, collection and aggregation are just part of the solution. Search capabilities allow you to find relevant logs to help troubleshoot production problems and monitor the health of different parts of your application. Many systems also include monitoring and configurable alerts to automatically identify problems and anomalies, and automatically alert support personnel to address them.

Comprehensive Alerting

The most common types of alerts identify error conditions within your application. More advanced systems like LogDNA allow you to specify the type of error and its frequency as part of the alert. In addition to error states, you can also configure an alert based on the results of a predefined query against your logs.

In addition to alerts based on the presence of an error condition or log query, you can also configure alerts based on the absence of certain types of logs. An example might be an online commerce site that expects a minimal level of transactions each hour. A lack of these transactions might indicate an access problem within the application.

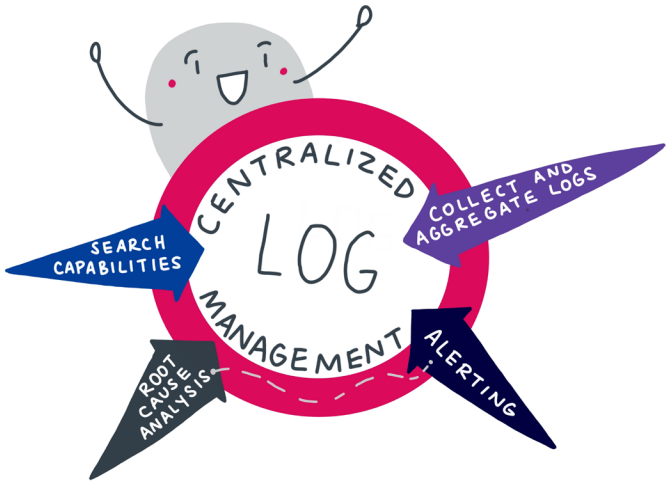
When used together, alerts for both the presence and absence of different conditions relieve your engineers of the responsibility to regularly review logs, and they can focus on building new features or improving their processes. You can learn more about alerting from the LogDNA Alerts Overview.

Root Cause Analysis

Once your support teams have received an alert from the log management system or a production incident has occurred, teams need to shift into resolution mode. The first step in the process is a root cause analysis. You can't resolve a problem until you have identified the reason it occurred and each of the components involved.

Identifying a problem in a distributed system can be incredibly challenging because you first need to identify which service is causing the problem and which services are affected by the situation. This investigation is where the ability to search through an aggregated collection of logs using request or transaction identifiers, time constraints, and additional filters is invaluable. You can learn more about how to search logs in the LogDNA Search Documentation.

Once you've identified the problem and worked on a solution, you'll typically want to deploy the solution while actively monitoring the logs. LogDNA's Live Tail lets you monitor logs as they are received from your application, giving you real-time visibility into the new deployment status and allowing you to validate its success or failure. You can also use the log management system to aggregate the logs and perform searches after time has passed.



CONCLUSION

In this eBook, we've shown how to leverage logs and LogDNA to drive three key stages of the SDLC: Development, QA and staging, and production.

There's no doubt that logging (and LogDNA) can help optimize other SDLC stages that we haven't discussed here. You can use logs to assess functionality requirements and plan new features during the planning phase of the SDLC, for example. Likewise, logs can help teams during deployment to ensure that a new application release is deployed smoothly, or to help manage complex deployment patterns such as those that come with canary or A/B releases.

In other words, no matter which stage of the SDLC you help manage, or which challenges you face, logs are one key resource to help you do your job better. And in a world where teams are expected to deliver new application releases multiple times per week, or even per day, engineers need every insight and data point available to them to keep the delivery pipeline flowing smoothly.

logdna

Thank You

Sales Contact: outreach@logdna.com
Support Contact: support@logdna.com
Media Inquiries: press@logdna.com