logdna.com

logdna

LOGDNA EBOOK

Using LogDNA for QA and Staging

Logging during QA and Staging allows engineers to anticipate what will happen in production, and helps them get to the root cause of any problems they detect.

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 our series of eBooks, 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. This eBook is focused on logging during QA and Staging.

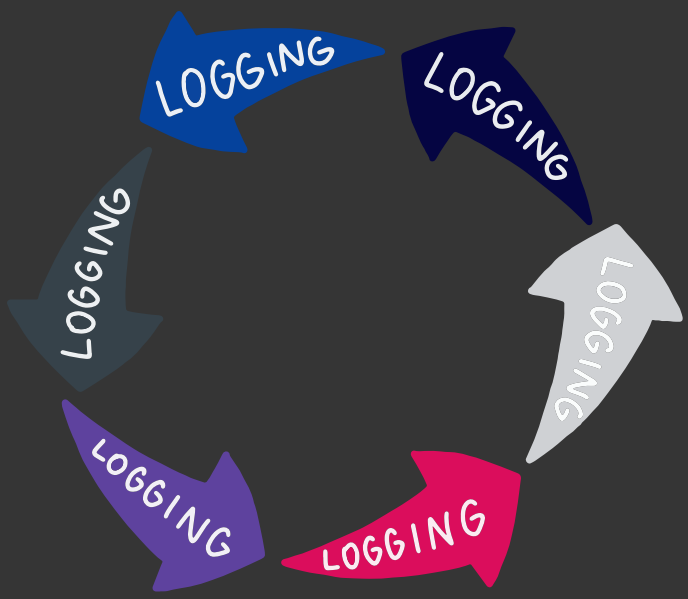


TABLE OF CONTENTS

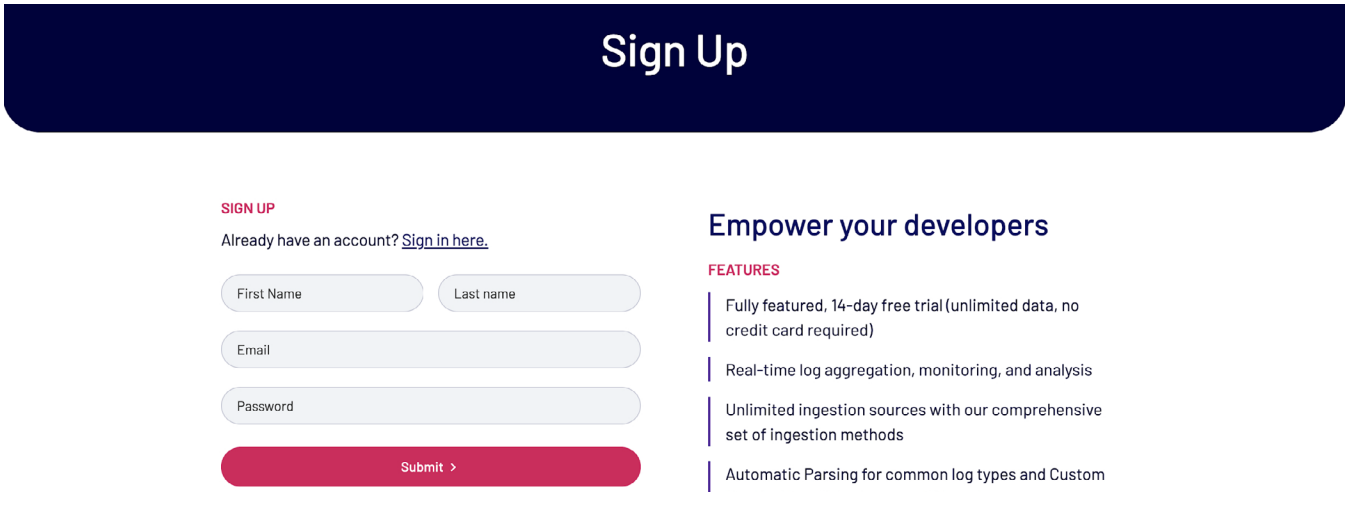
USING LOGDNA FOR QA AND STAGING

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.

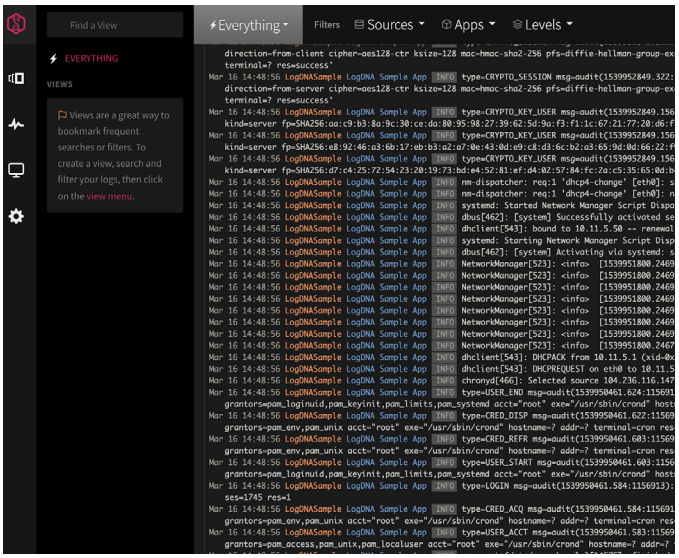
For QA and staging, then, logs must provide refined data that allows engineers to anticipate what will happen in production, and helps them get to the root cause of any problems they detect. This eBook explains how to use LogDNA for this purpose.

Setting Up A Staging Environment

The first thing you need to do is sign up. When this process is finished, you can explore the 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.

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:

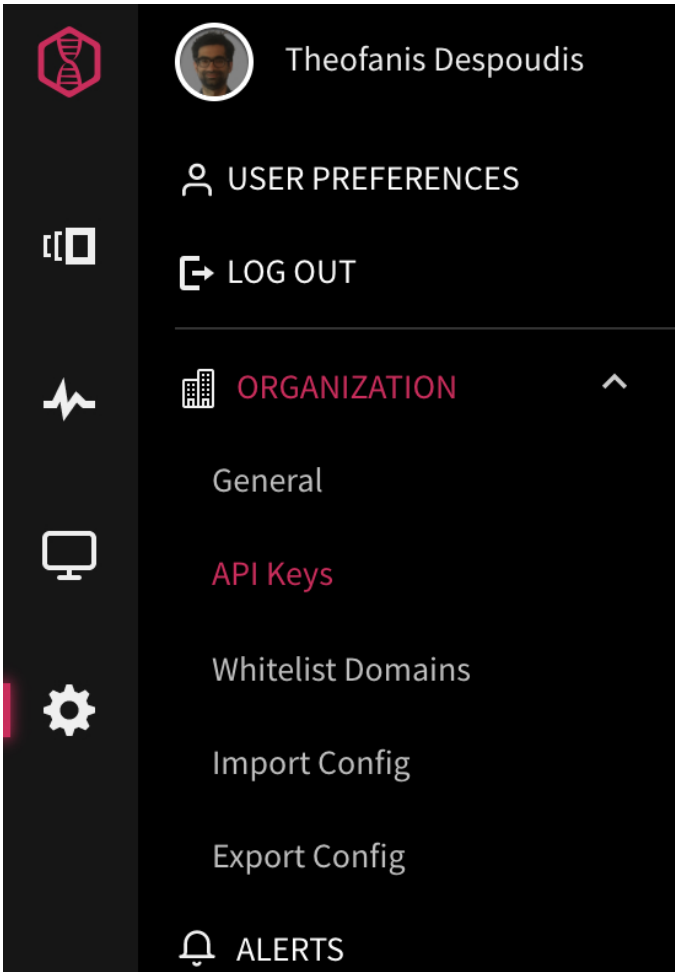
```
import winston from "winston";
import logdnaWinston from "logdna-winston";
import ip from "ip";
const logDNAOptions = {
   key: "LOGDNA_KEY",
   hostname: "localhost",
   ip: ip.address(),
   app: "Typescript-Node",
   env: "Development",
   indexMeta: true
};
const options: winston.LoggerOptions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

```
  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
```

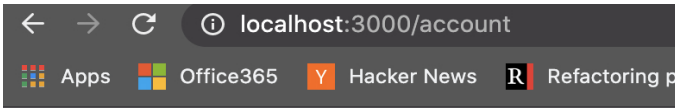
You will see the following events logged into the console:

```
(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);
});
```

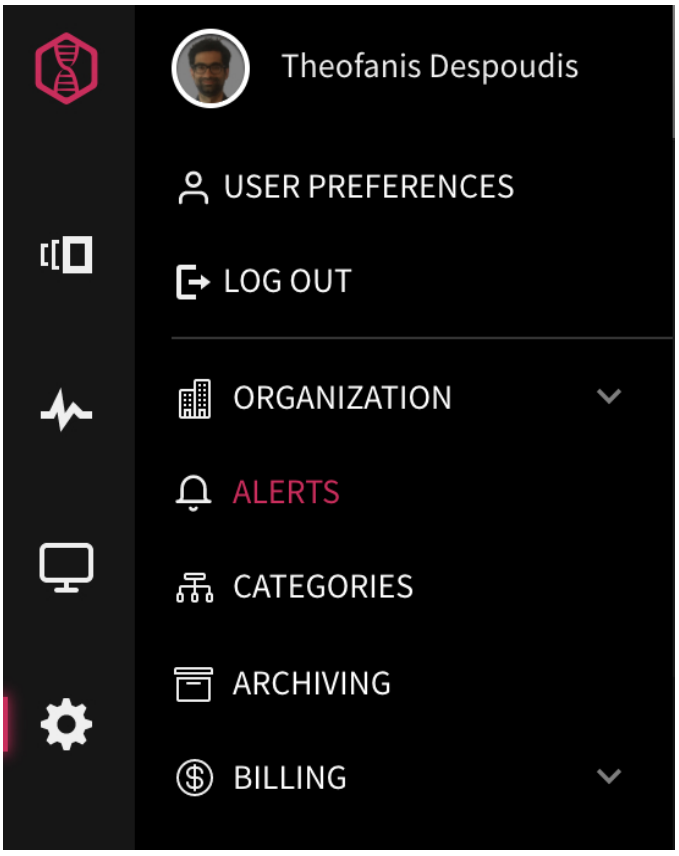
You may also want to capture access logs. You can add a **morgan** logger middleware in **`src/app.ts`**:

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

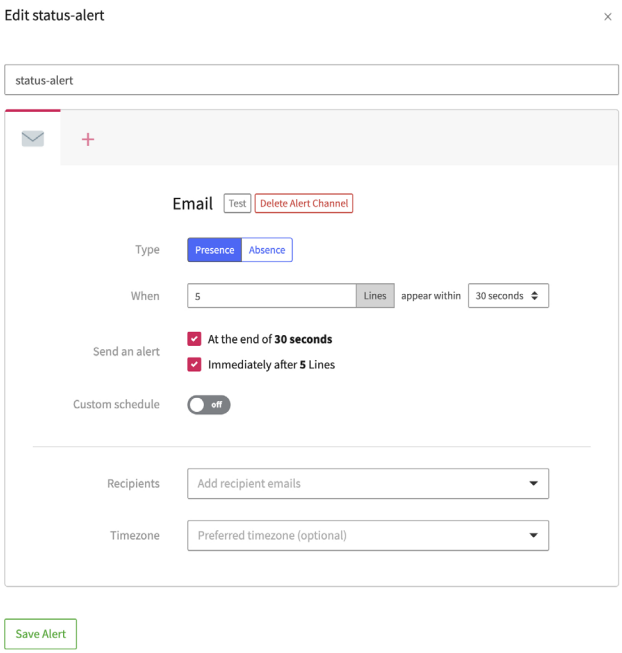
Now you can inspect the logs in the main Live Tail view:

Alerts

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

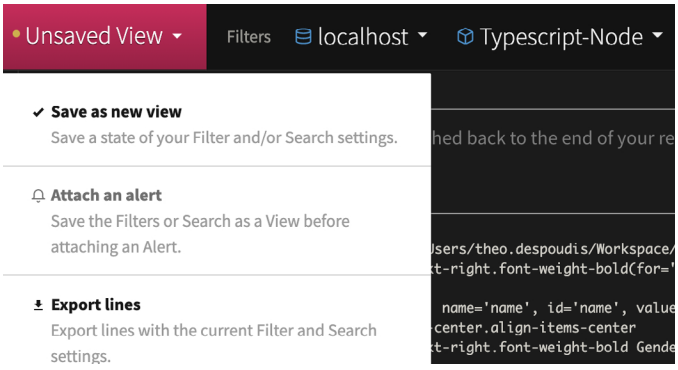


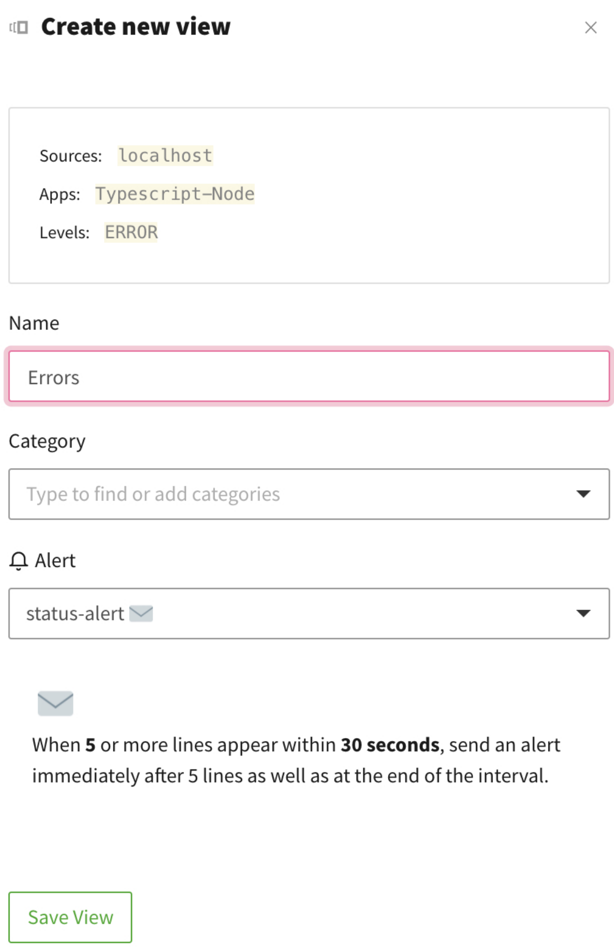
Let's first create an Alert that triggers an email after at least 5 events are logged. When you click 'Add Preset' you will need to fill in those details:



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.

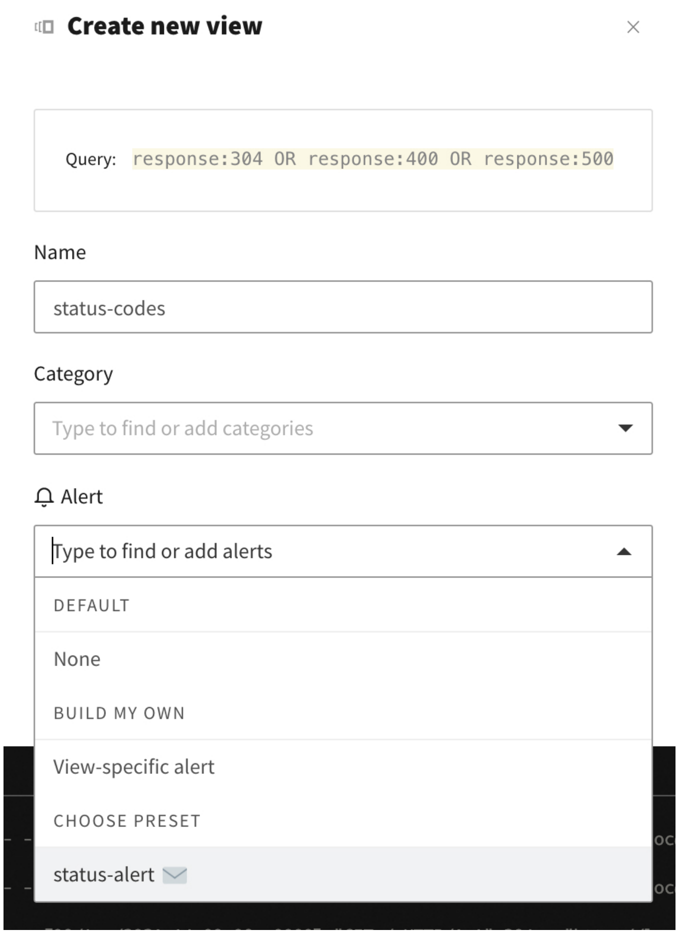




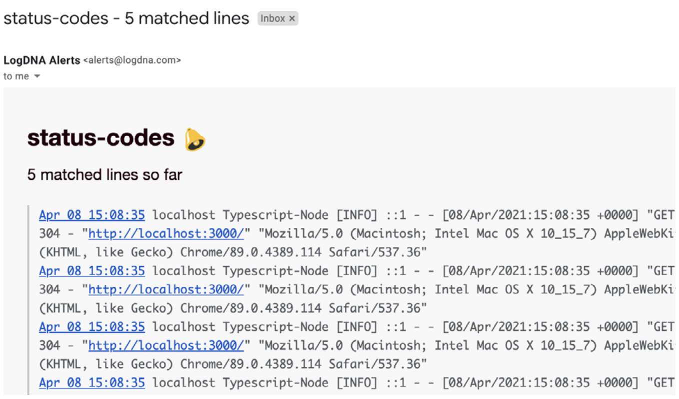
Let's add another one for specific status codes like 304, 400, and 500. You may want to use the bottom search bar. Enter the following query and save it as new View:

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

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

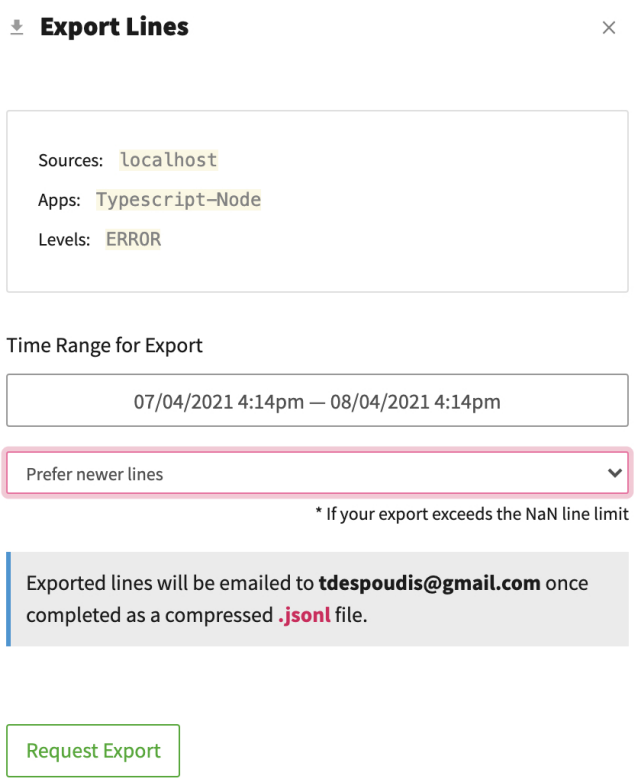


Now you will get email notifications when you exceed those alert thresholds:



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 .
```

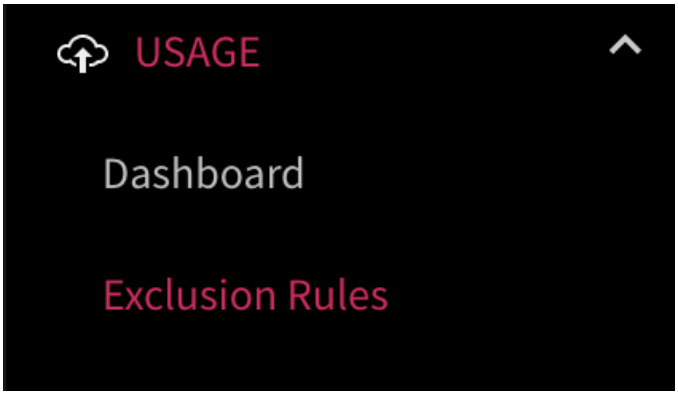
If the developers have access to the Dashboard, you can share the link to the View as well. This is an example link:

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:

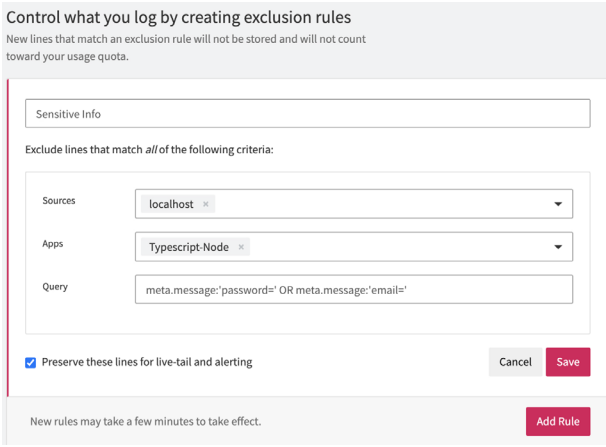


Then you will need to fill in some information about the rule. You may want to find specific log lines that match the query first. Here is an example with a message matching the following lines:

```
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 eBook, 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.

CONCLUSION

In this eBook, we've shown how to leverage logs and LogDNA during QA and Staging.

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