

DEVELOPER THINGS: JAVA LOGGING

A STACKIFY PUBLICATION

CONTRIBUTORS:
ERIC MARTIN
EUGEN PARASCHIV
MATT WATSON

EDITORS:
JENNILEE TANGPUZ
DOUG WARREN



TABLE OF CONTENTS

Introduction

Preface.....

Acknowledgements.....

44

Development

Logging Overview.....

Logging APIs.....

55

Basic Setup

Maven Configuration.....

Directory Structure.....

Initial Build.....

Local Maven Repository.....

SLF4J Logging API.....

Default Configuration.....

SimpleLogger Configuration.....

Log Levels.....

JSON Formatting.....

Conditional Logging.....

Parameterized Messages.....

Logging Exceptions.....

Mapped Diagnostic Contexts.....

681011121316171818

Logback

Logging Solutions.....

Logback Environment.....

Logback Example.....

212223

Spring Web Applications

Spring Initializr.....

Spring Maven Configuration.....

Greeting Object.....

Web Controller.....

Application Startup.....

Unit Tests.....

Spring Boot Application.....

Web Route URL Tests.....

Tomcat Access Log.....

252527282930313232

Logback Configuration	
Basic Default Configuration.....	33
Variable Substitution.....	34
Advanced Configuration.....	35
Appenders.....	36
Encoders and Layouts.....	38
Loggers.....	40
Filtering Logs.....	41
Conditional Configuration.....	45
Extending Logback.....	46
HTTP Request/Response Logging.....	47
Operations	
Getting Serious about Logging.....	51
Application Support.....	57
Work Smarter.....	57
Stackify Retrace	
Custom Logging Appenders.....	58
Direct Log4J12 Appender.....	58
Direct Logback Appender.....	58
Direct Log4J2 Appender.....	59
Data Masking.....	60
Standalone Logwatcher.....	60
Logging Dashboard.....	60
Log Management	
Centralized Cloud Solution.....	61
Filtering Logs.....	61
Examining Exceptions and Time Periods.....	61
Searching Your Logs.....	62
Exploring Java Exception Details.....	63
Application Monitoring	
Monitors & Alerts.....	65
Error Rates.....	65
Resolved Errors & New Errors.....	66
Log Monitors.....	66
Java Logging Best Practices.....	67
Conclusion	
Some Closing Words from Stackify.....	69





INTRODUCTION

ABOUT YOUR JAVA LOGGING GUIDE

This Java Guide provides best practices and tips for logging. We emphasize an integrated DevOps process that spans application development and operations. It consolidates, integrates, refines and extends content from the following previous Stackify blog articles.

- Solving Your Logging Problems with Logback
- Java Best Practices for Smarter Application Logging and Exception Handling

ACKNOWLEDGEMENTS

We thank the original authors of the source articles written for the Stackify blog, Eugen Paraschiv and Eric Martin. Also this could not have been accomplished without the significant effort from our technical editor, Doug Warren.

Furthermore, Jennilee Tangpuz made this project possible with support from Stackify Founder and CEO, Matt Watson, assistance from Bruce Solomon, and helpful reviews by Eric Martin, Natalie Sowards, and Darin Howard.





AN OVERVIEW

Initially, we focus on application development, testing, and debugging. We will cover the Simple Logging Facade for Java (SLF4J) API using its SimpleLogger implementation, and describe features and techniques in several examples including a Spring web application. Maven provides the build system with dependency management using a Java 8 platform with JUnit 5 testing.

We introduce the Logback logging framework, and explore its enhanced capabilities and advanced configuration with appenders, encoders, layouts, and loggers. Then we will address key operations topics using Log4J with the Stackify Retrace cloud solution for log management and application monitoring to support applications after deployment into a production environment.

WHY LOGGING MATTERS

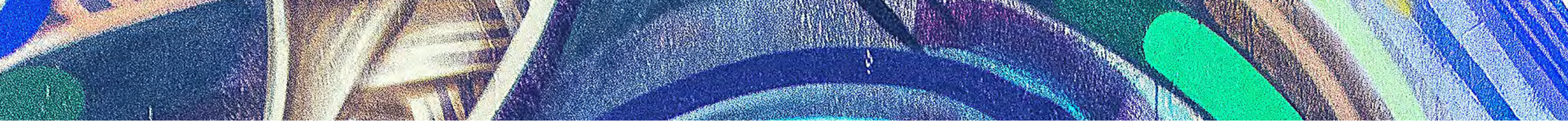
Logging is a crucial part of any application for both debugging and audit purposes, and choosing an adequate logging library is a foundational decision for any project.

There are many Java logging frameworks and libraries available, and most developers use one or more of them every day. Two of the most common examples for Java developers are Logback and Log4J. They are simple, easy to use, and work great for developers.

However, we should be logging better by now! Basic log files are just not enough, and these Java best practices and tips will help you make the most of them! Naturally, raw logging is just one aspect of understanding and reacting to a running application, next to [monitoring errors](#), [log management](#) and other techniques that create a more holistic picture of your system. We will also highlight those areas as well.

LOGGER APIS

There are several Logger APIs available, which can get very confusing. For example, there's SLF4J, Logback, Log4J, Log4J2, JDK java.util.logging (JUL), Apache Java Commons Logging (JCL), and probably others. So how do you know which ones to use, especially since they can be used in different combinations? Well, we're going to make it easy for you.



Each of these frameworks have their own unique logging API, but you should use the Simple Logging Facade for Java (SLF4J) API for whatever logging framework you choose underneath for your specific implementation. This common interface offers the greatest flexibility since it enables you to substitute any logging implementation without having to change your code.

It can also be very tricky to get all the dependencies and configurations correctly setup, but we have provided examples so that it works for you. That way you can focus on your application and its logging, rather than fighting to get your environment established properly.

BASIC SETUP

So let's get started with our examples. First we define a Maven project with dependencies for SLF4J, SimpleLogger, and JUnit 5; later we replace SimpleLogger with Logback. Then we can create our initial HelloLogging example, where we set up our Logger and demonstrate using the API to log messages.

MAVEN CONFIGURATION

We use the Apache Maven 3 build system for project information, dependency management, and documentation. It will utilize the standard project directory structure, POM (project object model) configuration, and plugins for development goals such as compile, test, and package. Furthermore, the examples are supported by JUnit 5 tests. First we need to define a Maven project with this pom.xml configuration file, which identifies our dependencies on SLF4j, SimpleLogger, JUnit5, and Java 8. Explore [Maven](#) and [JUnit](#) for details about setting up your development environment.

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

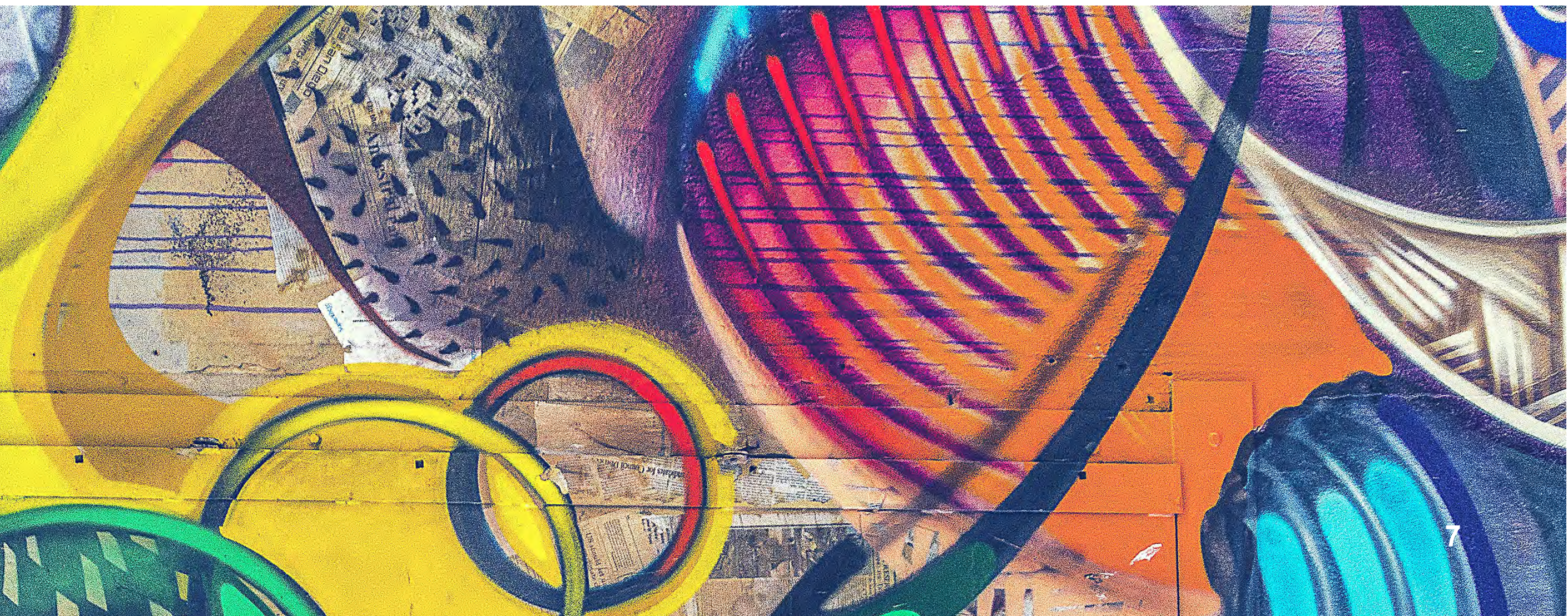
  <groupId>com.stackify.guide</groupId>
  <artifactId>logging-slf4j</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
```

```
<properties>
  <java.version>1.8</java.version>
  <maven.compiler.source>${java.version}</maven.compiler.source>
  <maven.compiler.target>${java.version}</maven.compiler.target>
  <file.encoding>UTF-8</file.encoding>
  <project.build.sourceEncoding>${file.encoding}
</project.build.sourceEncoding>
<project.reporting.outputEncoding>${file.encoding}
</project.reporting.outputEncoding>
</properties>
```

```
<dependencies>
  <!-- SLF4J: Simple Logging Facade for Java API -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.8.0-beta0</version>
  </dependency>
```

```
  <!-- SLF4J: SimpleLogger -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.8.0-beta0</version>
  </dependency>
```

```
  <!-- JUnit 5: Unit testing framework -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.0.2</version>
  </dependency>
</dependencies>
</project>
```



DIRECTORY STRUCTURE

Now create the main and test directory structure for Java source files and resources. You can do this in your IDE or issue these commands in a terminal.

```
mkdir -p src/main/java/com/stackify/guide/logging/slf4j/hello
mkdir -p src/main/resources
mkdir -p src/test/java/com/stackify/guide/logging/slf4j/hello
mkdir -p src/test/resources
```

It’s also suggested that you create a README.md file with [Markdown](#) text documentation for your project.

```
# logging-slf4j
Source examples for Java Guide: Logging (Stackify)
```

Your directories will now be ready for the package structure of the HelloLogging example.

```
~/Projects/stackify/guide/logging-slf4j
├── README.md
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── com
│   │   │   │   ├── stackify
│   │   │   │   │   ├── guide
│   │   │   │   │   │   ├── logging
│   │   │   │   │   │   │   ├── slf4j
│   │   │   │   │   │   │   │   └── hello
│   │   │   └── resources
│   └── test
│       ├── java
│       │   ├── com.stackify.guide.logging.slf4j
│       │   │   └── hello
│       └── resources
```

INITIAL BUILD

We haven’t even created any Java source files yet, but we have established a build system that we can use and extend throughout the project. So with the Maven project now defined, we can add the dependencies for our project and download the necessary JARs for our classpath by entering the following command in a terminal console, or by using the Maven support in your IDE.

```
mvn install
```

This compiles and tests any Java source files, copies resources, and packages the project artifact as a JAR archive file (logging-slf4j-1.0.0.jar).

At this point, our project directory structure has been extended with the compiled Java main and test classes used to package the JAR.

```
L--- target
  |-- classes
  |-- logging-slf4j-1.0.0.jar
  --- test-classes
```

LOCAL MAVEN REPOSITORY

Maven also installed that JAR and POM into the local Maven repository under the user's home directory (~/.m2/repository).

```
~/.m2/repository/com/stackify/guide
|-- logging-slf4j
|   |-- 1.0.0
|   |   |-- _remote.repositories
|   |   |-- logging-slf4j-1.0.0.jar
|   |   --- logging-slf4j-1.0.0.pom
|   --- maven-metadata-local.xml
```

SLF4J LOGGING API

The logging interface is very simple. You create one or more Loggers using a LoggingFactory, and then write messages with log statements for a logger at a specific logging level.

First, we need a few imports that specify the packages required for the SLF4J API that we are using. Of course, you could use a wildcard with a single statement **import org.slf4j.***; Although IDEs offer assistance to generate imports, it's important that the correct Logger is being selected so we make this explicit. For example, if you had multiple libraries in your classpath, you might get prompted for many different Logger classes in various packages.

Then, a Logger is created from the SLF4J LoggerFactory class. This example uses a single static Logger constant called log that will be initialized once for each class with the fully-qualified class name (com.stackify.guide.logging.slf4j.hello.HelloLogging). We're going to use "log" to produce our messages, rather than calling it "logger" (or LOGGER or LOG), since it seems to read better that way.

SIMPLELOGGER CONFIGURATION

Use this initial configuration in a properties file for the SLF4J SimpleLogger implementation. Save it as *simplelogger.properties* in the *src/main/resources* directory, which will be copied by Maven when packaging the project JAR (or WAR) artifact.

```
org.slf4j.simpleLogger.defaultLogLevel = TRACE
```

```
org.slf4j.simpleLogger.logFile = System.out  
# org.slf4j.simpleLogger.logFile = target/simple.log
```

```
org.slf4j.simpleLogger.showDateTime = true  
org.slf4j.simpleLogger.dateTimeFormat = yyyy.MM.dd HH:mm:ss z
```

```
org.slf4j.simpleLogger.showThreadName = false  
org.slf4j.simpleLogger.levelInBrackets = true  
org.slf4j.simpleLogger.showShortLogName = true
```

With `defaultLogLevel=TRACE`, it will include all log levels displayed on the console (`logFile=System.out`); however you can change this if you want it stored in a file instead, such as *target/simple.out* (as shown in the comment line).

The `showDateTime=true` property displays date/time with the `dateTimeFormat` based on Java [SimpleDateFormat](#); this example includes date with 4-digit year (yyyy), 2-digit month (MM), and 2-digit day (dd) separated by periods, as well as time with 2-digit hour (HH), 2-digit minutes (mm), 2-digit seconds (ss) separated by colon.

The timestamp is followed by timezone (CST for me). This might not be that useful in this simple starter configuration. However this becomes extremely important when you start aggregating logs (like at Stackify), especially from multiple applications on various platforms in different locations. In that case, it would probably be better to be GMT time in ISO8601 format (X), but we will use local time (CST) for now.

If you want more precise timing, you can add milliseconds (.S). The log level is shown in brackets (`levelInBrackets=true`). If you prefer WARNING (like Maven uses) for the WARN level, just override the default **warnLevelString**.

The default includes the fully-qualified logger name (`showLogName=true`), which would be `com.stackify.guide.logging.slf4j.hello.helloLogging` for this example.

However, you can choose to just display the last component of the logger name (usually the class, like `HelloLogging`) with `showSortLogname=true`. The latter is much simpler, except for complex applications, so we have selected that option. This will produce log output with the following format.

```
yyyy.mm.dd hh:mm:ss z [level] shortLogName message
```

So you now see the following output.

```
2018.01.11 15:40:24 CST [INFO] HelloLogging - Welcome to logging with the SLF4j API.
```

LOG LEVELS

The API makes the following 5 log levels available, arranged as a hierarchy in the following order. Errors (ERROR) are highest, then warnings (WARN) and information (INFO) Note that logging at INFO level would also include those above it (WARN and ERROR). The lowest levels are intended to provide finer (DEBUG) and finest (TRACE) details for developers. You can also use OFF in your configuration to suppress all logging output, but that might be dangerous since you would not see any errors or warnings.

- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Each of these levels has a corresponding logging method: *warn*, *error*, *info*, *debug*, and *trace*.

```
log.error("Action required!");  
log.warn("Might be a possible problem.");  
log.info("Useful in production logs.");  
log.debug("Information for developers.");  
log.trace("Log everything with details!");
```

Always use the logging level that provides the *smallest level of detail that is sufficient*. This reduces as much as possible the amount of logging data that you have to sift through. For example, with `defaultLogLevel=TRACE` in `simplelogger.properties`, you will see this output.

2018.01.06 15:14:52.826 CST [ERROR] Action required!
2018.01.06 15:14:52.826 CST [WARN] Might be a possible problem.
2018.01.06 15:14:52.826 CST [INFO] Useful in production logs.
2018.01.06 15:14:52.826 CST [DEBUG] Information for developers.
2018.01.06 15:14:52.826 CST [TRACE] Log everything with details!

It might be reasonably assumed for production applications that operations would always set log level at INFO to capture error, warning, and information messages; might sometimes use DEBUG for diagnostic investigation of intermittent problems; and never use TRACE because of potential performance implications. If you change `defaultLogLevel` to INFO, then you would only see this subset with lower-level TRACE and DEBUG messages suppressed.

2018.01.11 19:31:56 CST [ERROR] HelloLoggingTest - Action required!
2018.01.11 19:31:56 CST [WARN] HelloLoggingTest - Might be a possible problem.
2018.01.11 19:31:56 CST [INFO] HelloLoggingTest - Useful in production logs.

JSON FORMATTING

You might want to include objects formatted as JSON strings in your log messages, especially during debugging. There are various ways to read/write JSON and serialize arrays and objects; we will introduce a few common solutions with Gson (Google JSON) and Jackson.

Gson

This can be done easily using a Google JSON library called [Gson](#). Simply add the following to your Maven project configuration file (pom.xml) so that the JAR is added to your classpath.

```
<!-- Gson: Google JSON utility -->

<dependency>

    <groupId>com.google.code.gson</groupId>

    <artifactId>gson</artifactId>

    <version>2.8.2</version>

</dependency>
```

We will use a simple `Employee` class for our JSON examples under `src/main/java` having a `String` `name` property initialized by a public constructor and accessed with typical getter/setter methods.

```
package com.stackify.guide.logging.slf4j.employee;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Employee {
    private static final Logger log = LoggerFactory.getLogger(Employee.class);
    private String name;
    public Employee(String name) {setName(name);}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public String logString() {
        return "Employee (logString) {" + "name=" + name + "\" + "}";
    }
}
```

Then create a `JSONTest` class under `src/test/java` that demonstrates using `Gson` to log `Employee` objects as JSON messages in either a flat string or hierarchical pretty format at the debug level.

```
package com.stackify.guide.logging.slf4j.employee;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.junit.jupiter.api.Test;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

class JSONTest {
    static final Logger log = LoggerFactory.getLogger(JSONTest.class);

    Employee employee = new Employee("John Doe");

    @Test void testLogString() {
        log.debug(employee.logString());
    }
}
```

```

@Test
void testGsonMessage() {
    Gson gson = new Gson();
    log.debug("Employee (Gson object): {}", gson.toJson(employee));
}

@Test
void testGsonPretty() {
    Gson gsonPretty = new GsonBuilder().setPrettyPrinting().create();
    log.debug("Employee (Gson pretty): \n{}", gsonPretty.toJson(employee));
}
}

```

Running JSONTest shows the following console output.

```

2018.01.11 20:58:00 CST [DEBUG] JSONTest - Employee (logString) {name='John Doe'}
2018.01.11 20:58:01 CST [DEBUG] JSONTest - Employee (Gson object):
{"name":"John Doe"}
2018.01.11 20:58:01 CST [DEBUG] JSONTest - Employee (Gson pretty):
{
  "name": "John Doe"
}

```

Jackson

Another popular JSON formatting library is [Jackson 2](#), which uses an ObjectMapper along with [annotations](#) for [data binding](#). Add these dependencies to *pom.xml* to incorporate Jackson into the project.

```

<!-- Jackson 2 -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.3</version>
</dependency>

<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.9.3</version>
</dependency>

```

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.3</version>
</dependency>
```

Then add the necessary imports to JSONTest with a new JUnit 5 test for JSON by Jackson.

```
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
```

```
ObjectMapper mapper = new ObjectMapper();
```

```
@Test
```

```
void testJackson() throws JsonProcessingException {
    String result = mapper.writeValueAsString(employee);
    log.debug("Employee (Jackson): {}", result);
}
```

Testing JSONTest now adds this additional output for the new test.

```
2018.01.11 22:22:36 CST [DEBUG] JSONTest - Employee (Jackson): {"name":"John Doe"}
```

CONDITIONAL LOGGING

You can check whether a specific level is enabled before logging. For example, this would avoid string concatenation or building toString for a complex object argument.

```
if (log.isDebugEnabled()) {
    log.debug("Logger name: " + log.getName());
}
```

Running DevOps with NoOps.

Yes, it's possible. Here's how we do it.

In this podcast episode, hear how Stackify does DevOps to monitor their large production environment on Microsoft Azure - without an operations team.

**DEVELOPER
THINGS**

A STACKIFY ORIGINAL SERIES

[Click here to listen!](#)

PARAMETERIZED MESSAGES

In some cases, the log message may contain parameters which have to be evaluated. However, if the log level for the message is not enabled, then that is not really necessary.

For example, let's assume that we have an `Employee` class with a name, along with an `EmployeeService` that calculates a bonus for a given employee. However, if we were calculating bonuses for all our employees, we might not want that additional overhead unless we were debugging the application.

```
package com.stackify.guide.logging.slf4j.employee;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class EmployeeService {
    private static final Logger log = LoggerFactory.getLogger(EmployeeService.class);

    public int calculateBonus(Employee employee) {
        log.debug("Calculating bonus...");
        if (employee.getName().equals("Mary Smith")) {
            return 10000;
        }
        return 1000;
    }
}
```

One way to avoid this situation and therefore improve performance is to check the log level before logging the message and constructing the parameter:

```
Employee employee = new Employee("John Doe");
EmployeeService employeeService = new EmployeeService();
```

```
if (log.isDebugEnabled()) {
    log.debug("The bonus for employee: " + employee.getName() +
        " is " + employeeService.calculateBonus(employee));
}
```

As you can see, this is simple but can get a bit verbose. Another alternative is to make use of message format parameters where “{}” substitutes parameters only when the indicated level is enabled without having to surround it with an if condition block.

```
log.debug("The bonus for employee {} is {}", employee.getName(),
    employeeService.calculateBonus(employee));
```

This format ensures that the logger will first verify that the log level is enabled, and only afterward will use the value of the parameters to construct the log message. It is important to avoid such unnecessary method execution, serialization, and formatting.

There are method signatures for one or two parameters, as well as a special version for three or more variable arguments (...arguments); this last one does incur a minor cost of creating an Object[] array before invoking the method.

LOGGING EXCEPTIONS

It is critical to capture exceptions when errors happen so that you can identify the cause of problems and enable them to be resolved quickly. Besides the logger statements with a message and one or more object arguments, there is another variation shown here, where an exception is provided as the last argument.

```
try {
    // Database query: findEmployeeById(id)
} catch(SQLException ex) {
    log.error("Employee not found: {id}", id, ex);
}
```

Note that there is no placeholder for the exception like there is for id; also a message is required when you include an exception, even if it's only "Exception follows:". It's probably best to handle and log exceptions at the source, if possible.

Of course, you can choose to declare thrown exceptions, and handle them further up the stack where there might be more context, or a Thread.setDefaultUncaughtExceptionHandler. In any case, don't just catch and ignore, log and re-throw, or not include sufficient details about the error so it can be addressed.

MAPPED DIAGNOSTIC CONTEXTS

Logging helps to debug and audit complex distributed client-server or web applications and services that support multiple clients simultaneously, typically on separate threads. Of course, it would be hard to identify associated log output by user and/or transaction, so you might consider associating a separate logger for each client; however, this approach would be strongly discouraged because it proliferates loggers and their increased management overhead.

To address this requirement, the SLF4J logging API provides the [Mapped Diagnostic Context \(MDC\)](#) as a lighter technique to uniquely stamp each log request servicing a given client with contextual information, such as client id, IP address, and request parameters. The MDC is managed on a per-thread basis, and logging components will automatically include this information in each log entry. Also note that a child thread does not automatically inherit a copy of the mapped diagnostic context of its parent.

A simple static API enables you to put a diagnostic context String value as identified with a String key parameter into the current thread's diagnostic context map, which can be retrieved later by other logging components.

```
MDC.put("userRole", "ADMIN");
MDC.put("userId", "jdoe");
MDC.put("action", "Create.user");
```

```
log.info("Admin Action");
```

```
MDC.clear();
```

There is a special pattern layout in the encoder for the Logback configuration that you will see later. The `%mdc` or `%X` placeholder is used to format log messages with MDC context values associated with the thread that generated the event. In fact, there is an example of a sifting appender with a default MDC discriminator that will direct log messages to an ADMIN log based on a *userRole* key.

```
<pattern>%X{userRole} %X{userId} %X{action} - %m%n</pattern>
```

If the `mdc` conversion word is followed by a key between braces, as in `%mdc{userRole}`, then the MDC value corresponding to the key 'userRole' will be output. When the value is null, then the default value is output, if specified after the key with the `:-` operator, like `%mdc{userRole:-USER}`. If no default value is specified, then the empty string is output. If no key is given, then the entire content of the MDC will be output in the format "key1=val1, key2=val2".

For example, to produce `userRole=ADMIN, userId=jdoe, action=Create.user`, any of these `<pattern>` layout format conversion words would work.

```
userRole=%mdc{userRole}, userId=%mdc{userId}, action=%mdc{action}
userRole=%X{userRole}, userId=%X{userId}, action=%X{action}
userRole=%X{userRole:-USER}, userId=%X{userId}, action=%X{action}
```

However, **userRole=ADMIN, userId=jdoe, action=Create.user** would be produced by **%mdc** (or **%X**) alone, since the entire context map is output (with equals and comma delimiter). It might be good to use **%mdc** when you want the entire context map, and **%X{key}** to specify one or more individual values in the log message.

Actually, it might be preferable to convert a JSON object with key “json” to String as an MDC value. Then **%mdc** would produce **json={userRole=ADMIN, userId=jdoe, action=Create.user}**. However, if you still wanted to use **userRole** as a discriminator, you could use **%X{userRole}, %X{json}** to get **userRole=ADMIN, json={userId=jdoe, action=Create.user}**. There is a lot of flexibility here, but this emphasizes that you need to plan your logging system, and establishing standards would be very important.

You also have the ability to get or clear the context value for a specific key, as well as clear/set/get the entire context map (Map<String, String>). See the [MDC chapter](#) in the Logback manual and [Javadoc](#) for details.

This SLF4J API delegates to an underlying logging system with MDC functionality, which is only currently available with Logback and Log4J; an empty or basic adapter is provided for logging system without MDC support, such as SLF4J Simple Logger (or NOP), as well as JDK java.util.logging. Therefore, you can take advantage of this capability without dependencies on a particular logging implementation.





LOGBACK

LOGGING SOLUTIONS

Now that we understand how to use the common SLF4J application interface with the basic SLF4J SimpleLogger implementation, we can now explore the more advanced logging frameworks Logback and Log4J. First we cover the popular Logback logging framework for Java applications that was created as a successor to [Log4J 1.2](#).

Next we will create a web application using Spring Boot, which uses Logback for its default logging system, it also generates a lot of log messages from all the frameworks, like Spring WebMVC, Tomcat web server, JPA data persistence, Hibernate object/relational mapping (ORM), H2 relational database, SQL, REST web services, etc.

Then we will describe Apache Log4J 2, which also extended Log4J 1.2 with additional features beyond Logback. Finally, we illustrate some scenarios with Stackify Retrace cloud log management using Log4J 1.2.

Logback Advantages

These are some of the features and advantages of Logback over Log4J 1.2, which it supersedes. These highlight some reasons why Logback is a great choice for a logging framework.

- faster execution compared to Log4J 1.2
- native support of Simple Logging Facade for Java (SLF4J), which offers a common logging API that makes it easy to switch to a different logging framework later if necessary
- conditional processing of the defined configuration
- advanced filtering capabilities
- compression of archived log files
- support for setting a maximum number of archived log files
- HTTP access logging
- recovery from I/O failures

Logback Structure

The Logback project is organized into 3 main modules:

- logback-core: basic logging functionality
- logback-classic: additional logging improvements, such as SLF4J API support
- logback-access: integration with servlet containers, such as Tomcat and Jetty

LOGBACK ENVIRONMENT

To start using Logback requires adding the [logback-classic](#) dependency to the Java classpath. We do that in Maven by adding the Logback 1.2.3 dependency under `<dependencies>`. You will notice that we define Maven variables as `<properties>` so that we have all our software versions identified in one place, which makes it easier to upgrade when they change.

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.2.3</version>
</dependency>
```

This single dependency is enough since Maven will transitively pull in the additional *logback-core* and *slf4j-api* dependencies.

If no custom configuration for *logback.xml* is defined, Logback provides a simple automatic configuration. By default, this ensures that log statements are printed to the console at DEBUG level. Note that the SLF4J SimpleLogger default was INFO.

LOGBACK API

The Logback framework has its own native logging API, which uses a *LoggerContext* (instead of *LoggerFactory*) and a different *Logger*. Here's what logging looks like with that API.

```
import ch.qos.logback.classic.LoggerContext;
import ch.qos.logback.classic.Logger;
```

```
LoggerContext lc = new LoggerContext();
Logger log = lc.getLogger(UserServiceTest.class);
log.debug("Debug log message from native Logback API via LogContext");
```

LOG4J2 API

There is also a unique API for the Log4J2 framework; yet another Logger with a LogManager this time.

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

```
Logger log = LogManager.getLogger(UserServiceTest.class);
log.debug("Debug log message from native Log4J2 API via LogManager");
```

However, we are not going to use either of these. We suggest that you use the SLF4J API instead, which is supported by Logback, Log4J2, and Log4J 1.2, as well as SLF4J SimpleLogger.

LOGBACK EXAMPLE

This basic example for a user service shows logging using the same SLF4J API with our new Logback implementation. This demonstrates how different logging frameworks can be used without changing the developer interface.

UserService

Here is our user service class.

```
package com.stackify.logging.slf4j.user;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class UserService {
    private static final Logger log =
        LoggerFactory.getLogger(UserService.class);
    public UserService() {
        log.debug("New UserService created.");
    }
}
```

UserServiceTest

This is a JUnit 5 test for the user service.

```
package com.stackify.logging.slf4j.user;

import org.junit.jupiter.api.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```

public class UserServiceTest {
    private static final Logger log =
        LoggerFactory.getLogger(UserServiceTest.class);

    private UserService userService = new UserService();

    @Test
    public void testUserService() {
        log.debug("Testing UserService ...");
        // Test something!
    }
}

```

Test Logback Output

Since we now have Logback logging enabled, it's time to see it work by running the JUnit 5 tests with Maven again, which builds and tests everything in the project. Rather than using the console terminal, you will probably choose to use the enhanced IDE support available for Java, Maven, JUnit, and Spring to run the applications and tests.

```
mvn clean test
```

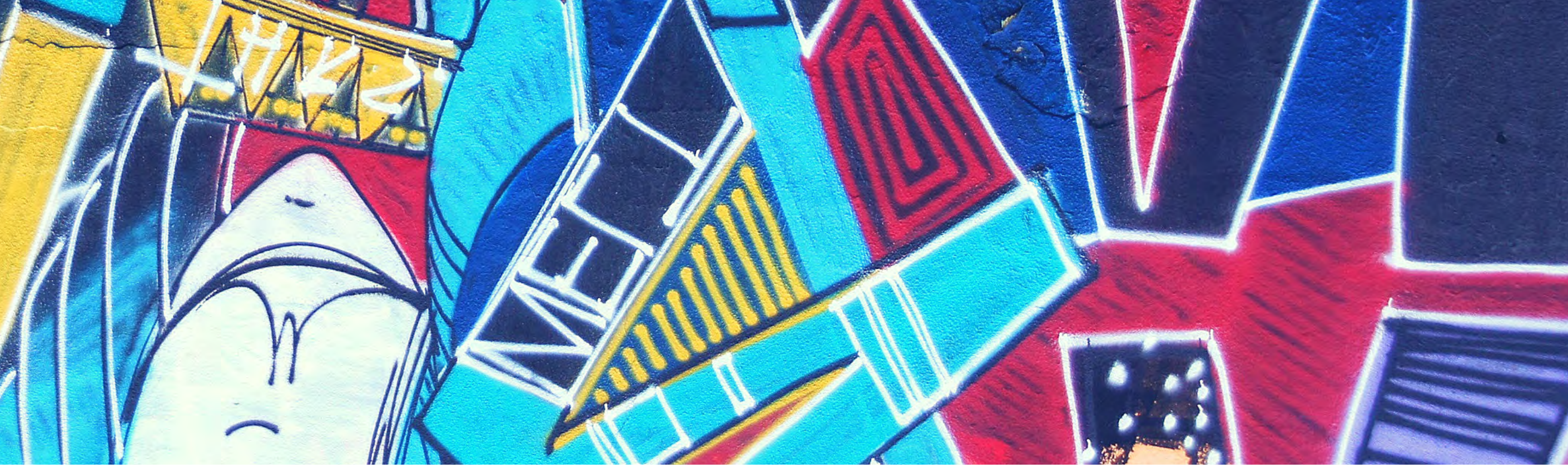
This is the output of running the tests.

```

22:55:08.159 [main] DEBUG com.stackify.logging.slf4j.user.UserService - New
UserService created.
22:55:08.166 [main] DEBUG com.stackify.logging..slf4j.user.UserServiceTest -
Testing UserService

```





SPRING WEB APPLICATIONS

Now let's create a web application using Spring Boot.

SPRING INITIALIZR

Using the [Spring Initializr](#) is the easiest way to create a Spring Boot project.

- Generate a Maven project with Java and Spring Boot 1.5.9
- Identify Project Metadata
 - Group: com.stackify.guide
 - Artifact: logging.spring
- Select Spring Boot Starters and dependencies
 - Web
- Generate Project: Downloads Maven project configuration file (pom.xml)
- Make these cosmetic adjustments
 - Change name from “demo” to “guide-logging-spring”
 - Update description to “Stackify Guide: Logging - Spring Boot Web app”
 - Change version from 0.0.1-SNAPSHOT to 0.0.1

SPRING MAVEN CONFIGURATION

We are now going to create a new project using this generated Maven POM project configuration file, which contains all the necessary starters and dependencies for a Spring Boot WebMVC application with an H2 database, and uses Logback as the default logging system.

```
<?xml version="1.0" encoding="UTF-8"/>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xs="http://www3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd"
  <modelVersion>4.0.0<modelVersion>
```

```
<groupId>com.stackify.guide</groupId>
<artifactId>logging-spring</artifactId>
<version>1.0.0</version>
<packaging>jar</packaging>
<relativePath/>

<name>logging-spring</name>
<description>
  Stackify Guide: Logging - Spring Boot Web app
</description>

<!-- Inherit Spring Boot Starter defaults from repository -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.9.RELEASE</version>
  <relativePath></relativePath>
</parent>

<dependencies>
  <!-- Spring Boot: Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- Spring Boot: Developer Tools -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
  </dependency>

  <!-- Spring Boot: Test -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

```

<build>
  <plugins>
    <!-- Package Spring Boot application as executable JAR -->
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${spring-boot.version}</version>
    </plugin>
  </plugins>
</build>

</project>

```

GREETING OBJECT

First, let's create a Greeting class with attributes of id (long) and content (String) with getters, as well a constructor for initialization. Also a toString method is provided for formatted string output (like log messages).

```

package com.stackify.guide.logging.spring.greeting;

public class Greeting {
    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() { return id; }
    public String getContent() { return content; }
    public String toString() { return "Greeting{" +
        "id=" + id + ", content=" + content + "\" + '\"';
    }
}

```

For more complex domain objects, you might want to provide a toLog method that produces a subset of all the attributes with an id and key information useful for application monitoring and log management. It can also be helpful to include a toJSON method that produces JSON-formatted output that could be used for structured logging.

WEB CONTROLLER

Next, we need to define a `GreetingController` as a REST web service controller for our Spring Web MVC application. This defines URL mappings for two routes. One (`/`) simply returns “Hello Spring”, and the other (`/greeting`) returns a custom greeting with an optional name request parameter (with default value of “Spring”) and produces a JSON document response using Jackson.

```
package com.stackify.guide.logging.spring.greeting;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import java.util.concurrent.atomic.AtomicLong;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
public class GreetingController {
    private static final Logger log =
        LoggerFactory.getLogger(GreetingController.class);

    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();

    @GetMapping("/")
    String home() {
        log.info("Hello Spring!");

        return "Hello Spring!";
    }

    @GetMapping("/greeting")
    Greeting greeting(
        @RequestParam(value = "name", defaultValue = "Spring") String name) {

        String response = String.format(template, name);
        long id = counter.incrementAndGet();
        Greeting greeting = new Greeting(id, response);

        log.info("{} ", greeting);

        return greeting;
    }
}
```

APPLICATION STARTUP

Then, of course, we must have the GreetingApp that starts our Spring Boot application.

```
package com.stackify.guide.logging.spring.greeting;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@SpringBootApplication
public class GreetingApp {
    private static final Logger log = LoggerFactory.getLogger(GreetingApp.class);

    public static void main(String[] args) {
        log.info("Starting Spring Boot REST WebMVC app...");

        SpringApplication app = new SpringApplication(GreetingApp.class);

        // Customize Spring Boot startup - No banner or startup messages
        app.setBannerMode(Banner.Mode.OFF);
        app.setLogStartupInfo(false);

        // Disable automatic restart and live reload.
        System.setProperty("spring.devtools.restart.enabled", false);

        app.run(args);
    }
}
```

This Spring Boot application (GreetingApp) is normally started with default setting by the static method `SpringApplication.run(GreetingApp.class)`, but here we create an instance to customize our startup before running it. We suppress the Spring Boot banner and turn off the startup INFO logs. Also we disable automatic restart and live reload (for now) to simplify the log output without the restartedMain thread. This illustrates how any system property can be programmatically set, although usually it would make more sense to use the `application.properties` external configuration file.

Normally, Spring Boot starts an embedded Tomcat server at port 8080. However, since you might already have another Tomcat server at that address, it might be best to define a different `server.port` like 8888 in the Spring Boot *application.properties* file under both the *src/main/resources* and *src/test/resources* directories.

```
server.port = 8888
```

UNIT TESTS

It would also be useful to have JUnit 5 tests for our Greeting, so let's add this `GreetingTest` class under `src/test/java`.

```
package com.stackify.guide.logging.spring.greeting;

import org.junit.jupiter.api.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class GreetingTest {
    private static final Logger log = LoggerFactory.getLogger(GreetingTest.class);

    @Test
    void testGreeting() {
        log.info("Starting Greeting tests...");
        Greeting greeting = new Greeting(1, "Hello");
        log.info(greeting.toString());
    }
}
```



SPRING BOOT APPLICATION

Finally, you can start the Spring web application via your IDE, or Maven command “mvn spring-boot:run”. You would normally see quite a few startup messages in the console until the GreetingApp is started in the embedded Tomcat web server on localhost:8888 (but we suppressed them).

```
:: Spring Boot ::      (v1.5.9.RELEASE)
```

```
2018.01.09 10:47:03 -06 [INFO] GreetingApp - Starting GreetingApp ...
...
2018.01.09 10:47:06 -06 [INFO] TomcatEmbeddedServletContainer - Tomcat
initialized with port(s): 8080 (http)
2018.01.09 10:47:06 -06 [INFO] StandardService - Starting service [Tomcat]
2018.01.09 10:47:06 -06 [INFO] StandardEngine - Starting Servlet Engine: Apache
Tomcat/8.5.23
2018.01.09 10:47:06 -06 [INFO] [/] - Initializing Spring embedded
WebApplicationContext
...
2018.01.09 10:47:07 -06 [INFO] TomcatEmbeddedServletContainer - Tomcat
started on port(s): 8080 (http)
2018.01.09 10:47:07 -06 [INFO] GreetingApp - Started GreetingApp in 4.15 seconds
(JVM running for 4.788)
2018.01.09 10:47:16 -06 [INFO] [/] - Initializing Spring FrameworkServlet
'dispatcherServlet'
2018.01.09 10:47:16 -06 [INFO] DispatcherServlet - FrameworkServlet
'dispatcherServlet': initialization started
2018.01.09 10:47:16 -06 [INFO] DispatcherServlet - FrameworkServlet
'dispatcherServlet': initialization completed in 26 ms
```

However, we have a much simpler log output, since we are now only logging WARN or ERROR messages from our application package, and ignoring INFO messages from Spring. You can suppress the startup banner with **spring.main.banner-mode=off** in *application.properties*.

```
15:47:37.671 [main] INFO com.stackify.guide.logging.spring.greeting.GreetingApp -
Starting Spring Boot REST WebMVC app via Tomcat on port 8888...
```

WEB ROUTE URL TESTS

Then try these scenarios in your browser.

1. localhost:8888

Hello Spring!

2. localhost:8888/greeting

`{"id":1,"content":"Hello, Spring!"}`

3. localhost:8888/greeting?name=John

`{"id":2,"content":"Hello, John!"}`

TOMCAT ACCESS LOG

Tomcat can produce an access log (similar to Apache or Nginx) that logs all incoming requests. The default embedded web server for Spring Boot has this disabled by default, but we will turn it on so you can see what the common log format looks like. We added these lines to `application.properties` to enable this access log feature and set the log directory (which must already exist). By default, these log files will have “`access_log`” prefix before the date (yyyy-MM) with `.log` suffix.

```
server.tomcat.accesslog.enabled = true
```

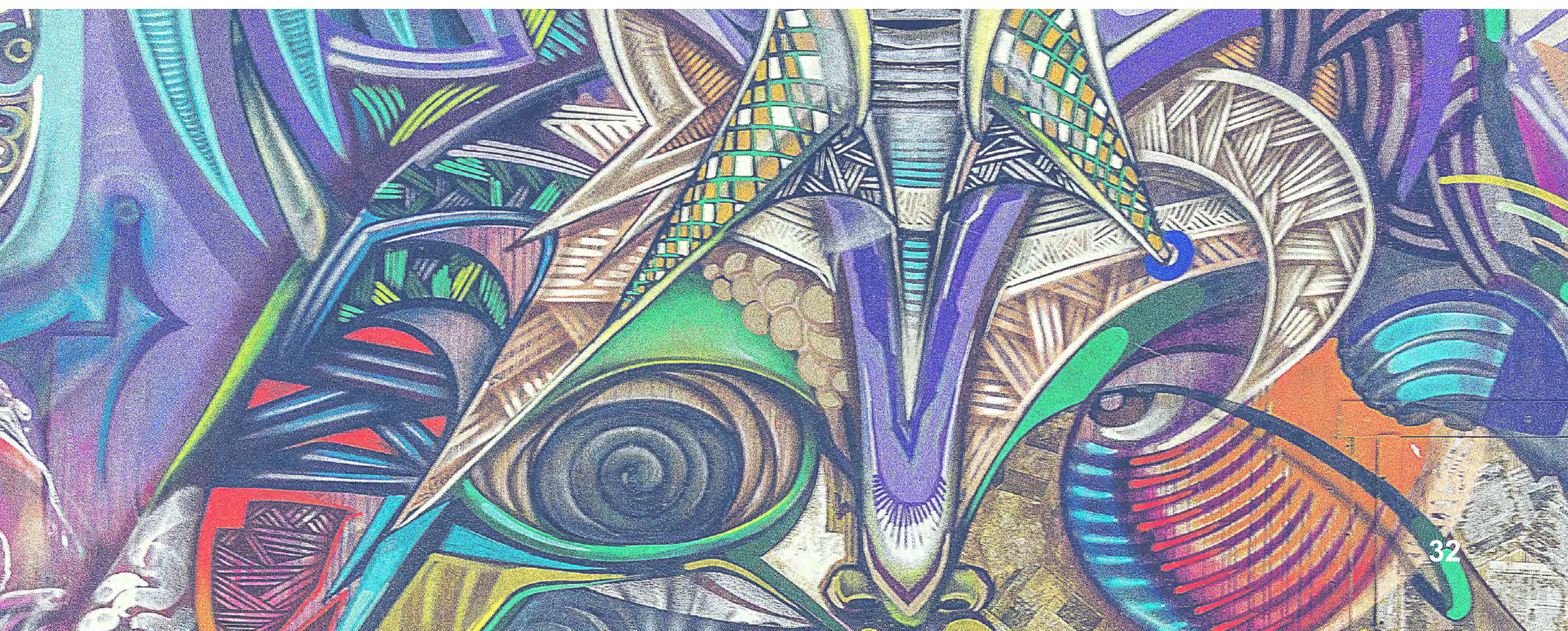
```
server.tomcat.accesslog.directory = /spring-tomcat-logs
```

Using `tail -f /spring-tomcat-lgs/access_log.2018.01.21.log` shows us the following requests.

```
0:0:0:0:0:0:0:0:1 - - [21/Jan/2018:16:14:51 -0600] "GET / HTTP/1.1" 200 13
```

```
0:0:0:0:0:0:0:0:1 - - [21/Jan/2018:16:14:46 -0600] "GET /greeting HTTP/1.1" 200 46
```

```
0:0:0:0:0:0:0:0:1 - - [21/Jan/2018:16:14:42 -0600] "GET /greeting?name=John  
HTTP/1.1" 200 44
```





LOGBACK CONFIGURATION

To create a configuration for Logback, you can use XML as well as Groovy. The system will automatically select and use the configuration, as long as you adhere to the naming convention.

There are three valid standard file names that you can choose from, which will be searched in this order:

- logback-test.xml
- logback.groovy
- logback.xml

BASIC DEFAULT CONFIGURATION

The examples in this tutorial will rely on a simple XML logback.xml file. Let's see what a basic configuration equivalent to the default one looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n</pattern>
    </encoder>
  </appender>
  <root level="INFO">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

This configuration defines a ConsoleAppender with a PatternLayout. Log messages on the console will be displayed at level DEBUG or above (INFO, WARN, ERROR) when using the defined pattern:

```
18:25:57.903 [main] INFO c.s.guide.logging.spring.greeting.GreetingApp - Starting
Spring Boot REST WebMVC app via Tomcat on port 8888...
```

An interesting and quite useful feature is that the library can automatically reload configuration files when they're modified. You can enable this behavior by setting the `scan="true"` attribute:

```
<configuration scan="true">
```

By default, the library scans and reloads files every minute, so it might be useful for long-running web applications.

To view the configuration log, you can add the `debug="true"` attribute:

```
<configuration debug="true">
```

This can also be quite helpful for development since it really speeds up identifying potential configuration errors.

VARIABLE SUBSTITUTION

Logback configuration files support defining and substituting variable values.

Simply put, variables can be defined using `<variable>` elements; however, `<property>` is also accepted since that's what variables were called in earlier versions (as well as Log4J), and you may see that in many configuration files or examples. Even though the Logback documentation emphasizes variable substitution, their own configuration examples still use `<property>`.

Variables can be defined inline within the configuration file like this:

```
<variable name="fileName" value="file.log">
```

Then you can access the variable using the typical `${}` syntax:

```
<file>${fileName}</file>
```

Another option is to define variables externally, where they are loaded from a properties file or classpath resource:

```
<variable resource="application.properties">
```

The properties defined in the *application.properties* file will be defined as variables in the *logback.xml* file.

Let's take a closer look at each of the main configuration elements to start putting together more complex, and ultimately more useful configurations.

ADVANCED CONFIGURATION

This section covers additional configuration features, such as appenders, encoders, layouts, loggers, filters, conditional processing, and extensions.

APPENDERS

In the Logback architecture, appenders are the elements responsible for writing log statements. All appenders must implement the Appender interface.

Furthermore, each appender corresponds to a certain type of output or mode of sending data. Here are some of the most helpful appenders that you can configure:

- *ConsoleAppender* – writes messages to the system console
- *FileAppender* – appends messages to a file
- *RollingFileAppender* – extends the *FileAppender* with the ability to roll over log files
- *SMTPAppender* – sends log messages in an email, by default only for ERROR messages
- *DBAppender* – adds log events to a database
- *SiftingAppender* – separates logs based on a runtime attribute

Let's see a few configuration examples for some of these.

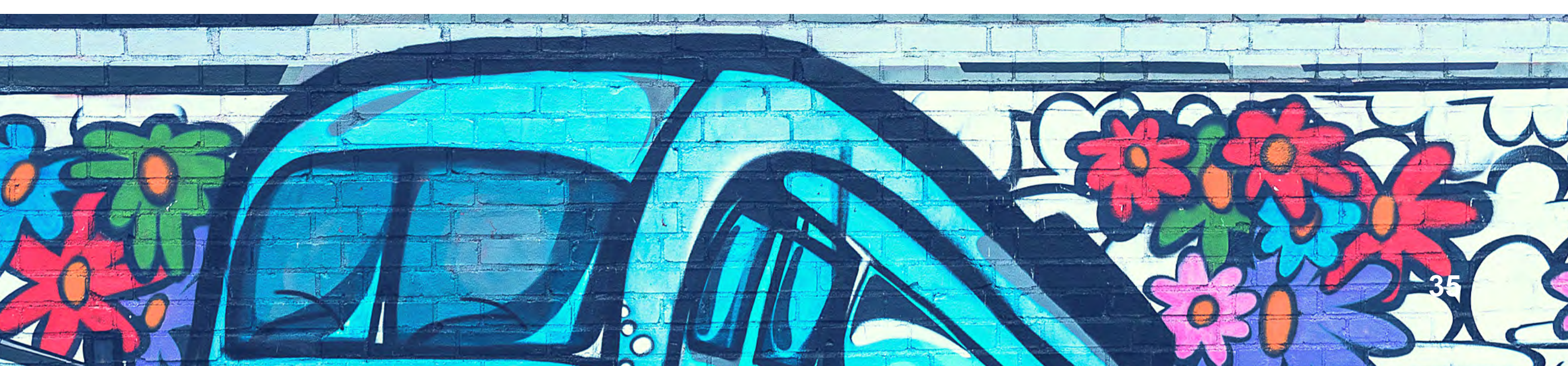
Standard Console Output

The *ConsoleAppender* is one of the more basic appenders available in Logback since it can only log messages to *System.out* or *System.err*.

The configuration for this appender usually requires specifying an encoder, as we saw in the basic example configuration from the previous section.

By default, messages are logged to *System.out*, but you can change that using the *target* attribute:

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
  <target>System.err</target>  
</appender>
```



Rollover Log Files

Of course, using a `FileAppender` for logging to a file is naturally the better way to go in any kind of production scenario where you need persistent logs. However, if all the logs are kept in a single file, this runs the risk of becoming too large and difficult to wade through. It's also makes long-term storage and warehousing of log data very difficult.

That's when rolling files come in handy. To address this well-known limitation, Logback provides the `RollingFileAppender`, which rolls over the log file when certain conditions are met (like time or size). For example this might be monthly, daily, or hourly; you might use archive folders (yyyy-MM), compress them as zip files, and choose your retention period.

The appender has two components:

- `RollingPolicy` – how rollover is performed
- `TriggeringPolicy` – when file is rolled over

To better understand these policies, let's create an appender which makes use of a *`TimeBasedRollingPolicy`* and a *`SizeBasedTriggeringPolicy`*:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n</pattern>
    </encoder>
  </appender>

  <appender name="rollingFileAppender"
    class="ch.qos.logback.core.rolling.RollingFileAppender">

    <rollingPolicy
      class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>log-%d{yyyy-MM-dd}.log</fileNamePattern>
        <maxHistory>30</maxHistory>
        <totalSizeCap>3GB</totalSizeCap>
      </rollingPolicy>

    <triggeringPolicy
      class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
        <maxFileSize>3MB</maxFileSize>
      </triggeringPolicy>
```

```

    <encoder>
      <pattern>[%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="INFO">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="rollingFileAppender" />
  </root>
</configuration>

```

The *TimeBasedRollingPolicy* implements both a *RollingPolicy* and a *TriggeringPolicy*.

The example above configures the `fileNamePattern` attribute based on the day – which means the name of each file contains the current date, and also that the rollover will happen daily.

Notice how we’re limiting the log data here – *maxHistory* is set to a value of 30, alongside a *totalSizeCap* of 3 GB – which means that the archived logs will be kept for the past 30 days, up to a maximum size of 3 GB.

Finally, the *SizeBasedTriggeringPolicy* configures the rollover of the file whenever it reaches 3 MB. Of course that’s quite a low limit, and a [mature log-viewing tool](#) like Retrace from Stackify can certainly handle a lot more than that.

You can now see how we’ve slowly moved out from basic examples to a more realistic configuration that you can actually start using as the project moves towards production.

User Context Logs

The *SiftingAppender* can be useful in situations when you want logs to be separated (or sifted) based on a runtime attribute such as the user session. For example, this might be used to create distinct log files per user, since it can separate logging events according to user session via dynamic nested appenders.

A class attribute on the `<discriminator>` tag identifies the type used for sifting (default, context, access, JNDI, or MDC). If none is specified, the default is *MDCBasedDiscriminator*, where the discriminating value is the MDC value associated with the given key property; if that MDC value is null, then the `defaultValue` is used.

For the discriminator to have access to the `userRole` key, you need to place it in the MDC (Mapped Diagnostic Context), which enables you to set information to be later retrieved by other Logback components using a simple static API:

```
MDC.put("userRole", "ADMIN");
log.info("Admin Action");
```

This will write the log message in a file called `ADMIN.log`, with all others going to the default `ANONYMOUS.log`.

To see this functionality in action, let's configure a *SiftingAppender* that separates logs into different files based on the *userRole* key, and add another `<appender-ref>` under the `root` logger:

```
<appender name="roleSiftingAppender"
  class="ch.qos.logback.classic.sift.SiftingAppender">
  <discriminator>
    <key>userRole</key>
    <defaultValue>ANONYMOUS</defaultValue>
  </discriminator>
  <sift>
    <appender name="fileAppender"
      class="ch.qos.logback.core.FileAppender">
      <file>${userRole}.log</file>
      <encoder>
        <pattern>%d [%thread] %level %mdc %logger{50} - %msg%n</pattern>
      </encoder>
    </appender>
  </sift>
</appender>
```

ENCODERS AND LAYOUTS

Now that you're starting to understand how appenders work, and just how flexible and powerful they are, let's focus on another foundational component in Logback.

The components responsible for transforming a log message to the desired output format are layouts and encoders.

Layouts can only transform a message into a *String*, while encoders are more flexible and can transform the message into a byte array and then write that to an *OutputStream*. This means encoders have more control over when and how bytes are written.

As a result, starting with version 0.9.19, layouts have been deprecated, but they can still be used for a transition period. If you do still use layouts actively, Logback will print a warning message:

This appender no longer admits a layout as a sub-component, set an encoder instead.

While they're starting to be phased out, layouts are still widely used and quite a powerful component on their own, so they're worth understanding.

Some of the most commonly used layouts are `PatternLayout`, `HTMLLayout`, and `XMLLayout` – let's have a quick look at these in practice.

The `PatternLayout` layout creates a `String` from a log message based on a conversion pattern, which is quite flexible and allows declaring several conversion specifiers; these can control the characteristics of the output `String` such as length and color, and can also insert values into the output `String`.

Let's see an example of a `PatternLayout` that prints the name of the logging thread in green, logger name with a length of 50 characters, and log levels using different colors with the `%highlight` modifier. Earlier versions of Logback wrapped a `<pattern>` in a `<layout>`, which has been deprecated and replaced by `<encoder>`; so the `PatternLayout` is now wrapped in a convenient `PatternLayoutEncoder` designed for this layout:

```
<appender name="colorAppender" class="ch.qos.logback.core.ConsoleAppender">
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%d %green([%thread]) %highlight(%level) %logger{50} -
%msg%n</pattern>
  </encoder>
</appender>
```

The output of this configuration shows log lines with the format we defined:

```
2018-01-21 19:09:36,829 [main] INFO
c.s.guide.logging.spring.greeting.GreetingApp - Starting Spring Boot REST WebMVC
app via Tomcat on port 8888...
```

Web Page Output

The `HTMLLayout` displays log data in an HTML table format, to which you can add custom styles. Let's configure an `HTMLLayout` using a `LayoutWrappingEncoder` to avoid the deprecation warnings:

```
<appender name="htmlAppender" class="ch.qos.logback.core.FileAppender">
  <file>log.html</file>
  <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
    <layout class="ch.qos.logback.classic.html.HTMLLayout">
      <pattern>%thread%level%logger%msg</pattern>
    </layout>
  </encoder>
</appender>
```

In the example above, the layout is used by a *FileAppender* to print logs to a *log.html* file. Here’s what the content of the HTML file will look like, using the default CSS.

Log session start time Mon Jan 22 08:51:41 CST 2018

Thread	Level	Logger	Message
main	INFO	com.stackify.guide.logging.spring.greeting.GreetingApp	Starting Spring Boot REST WebMVC app via Tomcat on port 8888...

So far, we’ve used the layout examples in the two main encoders available: *PatternLayoutEncoder* and *LayoutWrappingEncoder*. The purpose of these encoders is to support the transition from layouts to encoders.

LOGGERS

Loggers are the third main component of Logback, which developers can use to log messages at a certain level (and higher). Note that the levels may be specified as either lowercase (“debug” - like API methods) or uppercase (“DEBUG” - like static Level constants).

In previous examples, we’ve seen a configuration of the root logger:

```
<root level="debug">
  <appender-ref ref="STDOUT" />
</root>
```

A root logger at the top of the logger hierarchy is always provided, even if you don’t configure it explicitly; if none is defined, it will use a default *ConsoleAppender* at the *DEBUG* level. However, our example explicitly configures a root logger at *DEBUG* level using the *STDOUT* console appender defined earlier.

Let’s now define another logger, with an *INFO* level, which uses the *rollingFileAppender*:

```
<logger level="info" name="rollingFileLogger">
  <appender-ref ref="rollingFileAppender" />
</logger>
```

If you don't explicitly define a log level, the logger will inherit the level of its closest ancestor; in this case, the DEBUG level of the root logger.

As you can see, the name attribute specifies a logger name that you can later use to retrieve that particular logger:

```
Logger rollingFileLog = LoggerFactory.getLogger("rollingFileLogger");
rollingFileLog.info("Testing rolling file log");
```

What's interesting here is that you can actually also configure the log level programmatically, by casting to the *ch.qos.logback.classic.Logger* class, instead of the *org.slf4j.Logger* interface:

```
ch.qos.logback.classic.Logger rollingFileLog =
    (ch.qos.logback.classic.Logger)
        LoggerFactory.getLogger("rollingFileLogger");
rollingFileLog.setLevel(Level.DEBUG);
rollingFileLog.debug("Testing rolling log level");
```

Logger Additivity

By default, a log message will be displayed by the logger that writes it, as well as ancestor loggers. And since root is the ancestor of all loggers, all messages will also be displayed by the root logger.

To disable this behavior, you need to set the *additivity*=*"false"* property on the *logger* element:

```
<logger level="info" name="rollingFileLogger" additivity="false">
  ...
</logger>
```

FILTERS

Deciding what log information gets processed based on the log level is a good way to get started, but at some point, that's simply not enough.

Logback has solid support for additional log filtering, beyond just the log level. This is done with the help of filters – which determine whether a log message should be displayed or not.

Simply put, a filter needs to implement the *Filter* class, with a single *decide()*

method. This method returns enumeration values of type *FilterReply*: *DENY*, *NEUTRAL* or *ACCEPT*.

The *DENY* value indicates the log event will not be processed, while *ACCEPT* means the log event is processed, skipping the evaluation of the remaining filters.

Finally, *NEUTRAL* allows the next filters in the chain to be evaluated. If there are no more filters, the message is logged.

Here are the primary types of filters we have available: *LevelFilter*, *ThresholdFilter* and *EvaluatorFilter*.

Filtering Levels

The *LevelFilter* and *ThresholdFilter* are related to the log level, with the difference that *LevelFilter* verifies if a log message is equal to a given level, while the *ThresholdFilter* checks if log events are at or above a specified level.

Let's configure a *LevelFilter* that only allows ERROR messages:

```
<appender name="STDOUT_LEVEL_FILTER_APPENDER"
class="ch.qos.logback.core.ConsoleAppender">
  <filter class="ch.qos.logback.classic.filter.LevelFilter">
    <level>ERROR</level>
    <onMatch>ACCEPT</onMatch>
    <onMismatch>DENY</onMismatch>
  </filter>
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
  </encoder>
  <target>System.err</target>
</appender>
```

As you can see, the filter is associated with an appender that outputs the messages to the *System.err* target.

Filtering Thresholds

Similarly, you can configure the `ThresholdFilter` by specifying the `level` attribute below which the filter rejects messages. This filters out TRACE (below DEBUG), but accepts DEBUG and those above (INFO, WARN, and ERROR).

```
<appender name="STDOUT_THRESHOLD_FILTER_APPENDER"
class="ch.qos.logback.core.ConsoleAppender">
  <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
    <level>DEBUG</level>
  </filter>
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
  </encoder>
</appender>
```

Conditional Filters

Now let's have a look at the *EvaluatorFilter*, which we can use for more complex conditions. The *EvaluatorFilter* implements the same *decide()* method as the two level-based filters above, and uses an *EventEvaluator* object to determine whether a log message is accepted or denied.

There are actually two implementations available:

- *GEventEvaluator* – contains a condition written in Groovy
- *JaninoEventEvaluator* – uses a Java expression as an evaluation condition

Both evaluators require additional libraries on the classpath: groovy-all for the first *EventEvaluator* and janino for the second.

Let's take a look at how to define a Java-based *EventEvaluator*.

First, you need the *janino* dependency:

```
<dependency>
  <groupId>org.codehaus.janino</groupId>
  <artifactId>janino</artifactId>
  <version>3.0.8</version>
</dependency>
```

The evaluation condition has access to several objects, including event, message, logger, and level. Based on these, you can configure a filter using a *JaninoEventEvaluator*:

```

<appender name="STDOUT_EVALUATOR_FILTER_APPENDER"
  class="ch.qos.logback.core.ConsoleAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="ch.qos.logback.classic.boolex.JaninoEventEvaluator">
      <expression>
        return(level > DEBUG & &
          message.toLowerCase().contains("employee"));
      </expression>
    </evaluator>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
  </encoder>
</appender>

```

The example above configures a filter that only accepts log messages that have a level higher than DEBUG and contain the “employee” text.

For more high-level filtering, Logback also provides the TurboFilter class.

Limit Duplicate Messages

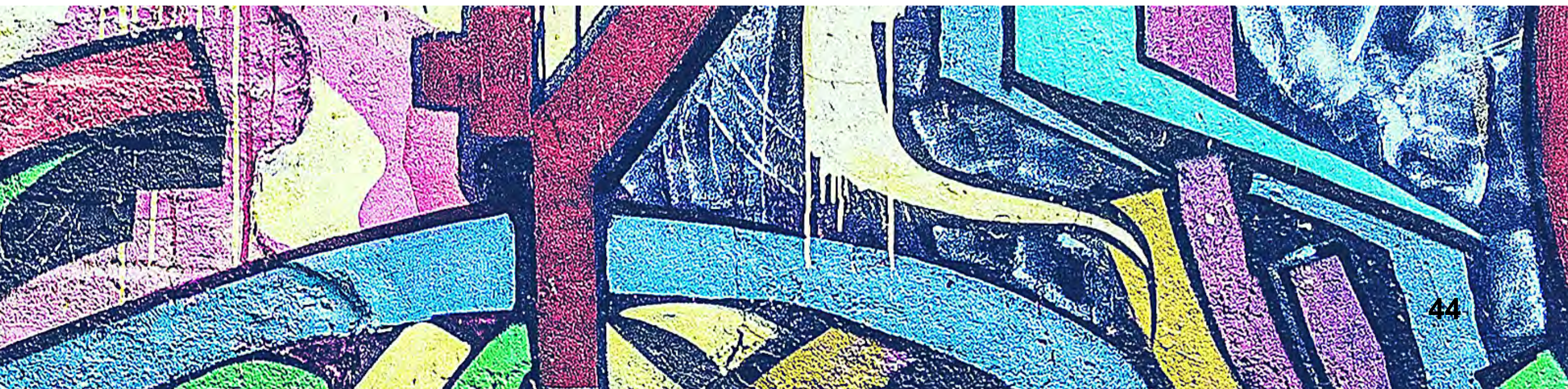
The TurboFilter filter behaves in a similar way to the Filter class, with the distinction that it’s not associated with a specific appender. Instead of accessing a logger object, it’s connected to the logging context and is invoked for every logging request.

Here’s a simple implementation of this class – the DuplicateMessageFilter. This configuration only allows 2 repetitions of the same log message (meaning 3 instances of it) and eliminates all subsequent ones.

```

<configuration>
  <turboFilter
    class="ch.qos.logback.classic.turbo.DuplicateMessageFilter">
    <AllowedRepetitions>2</AllowedRepetitions>
  </turboFilter>
</configuration>

```



CONDITIONAL CONFIGURATION

Sometimes you want to use different configuration file options for various situations or environments based on specific conditions.

Logback supports `<if>`, `<then>`, `<else>` elements that control whether a part of the configuration is processed or not. This is a unique feature among logging libraries and requires the previously mentioned *janino* library.

To define the conditions evaluated to control the processing of configuration, you can use the Java language. Furthermore, the expressions only have access to context or system properties.

A common use case is enabling a different configuration for separate environments:

```
<property scope="context" resource="application.properties" />
<if condition='property("env").equals("dev")'>
  <then>
    <root level="TRACE">
      <appender-ref ref="STDOUT" />
    </root>
  </then>
</if>
```

This example configures the root logger to display messages of all levels to the console, but only for the development environment, defined through an `env=dev` property in the *application.properties* file.

EXTENDING LOGBACK

Beyond the many features that Logback already contains, its architecture allows for the possibility of creating custom components that you can use in the same way as the default ones.

For example, here are several ways you can extend Logback functionality by creating custom elements.

- appender - extend *AppenderBase* and implement *append()*
- layout - subclass *LayoutBase* and define *doLayout()*
- filter - extend *Filter* and implement *decide()*
- *TurboFilter* - extend *TurboFilter* and override *decide()*

The configuration of custom elements is the same as standard elements.

Let's define a custom *TurboFilter* that will ignore all log messages of a specific logger:

```
package com.stackify.guide.logging;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.Logger;
import ch.qos.logback.classic.turbo.TurboFilter;
import ch.qos.logback.core.spi.FilterReply;
import org.slf4j.Marker;

public class IgnoreLoggerFilter extends TurboFilter {
    private String loggerName;

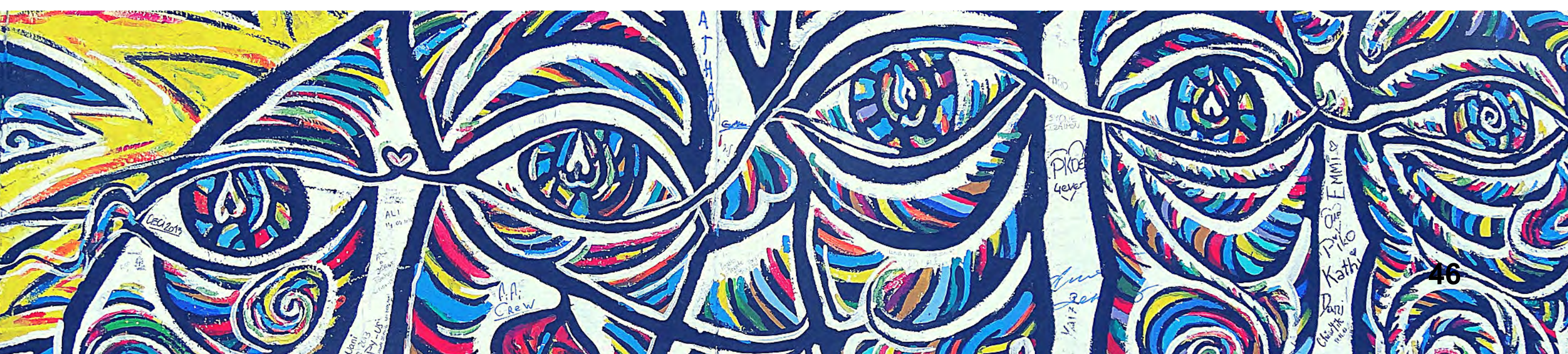
    @Override
    public FilterReply decide(Marker marker, Logger logger,
        Level level, String format, Object[] params, Throwable t) {
        if (loggerName == null) {
            return FilterReply.NEUTRAL;
        } else if (loggerName.equals(logger.getName())) {
            return FilterReply.DENY;
        } else
            return FilterReply.NEUTRAL;
    }

    public void setLoggerName(String loggerName) {
        this.loggerName = loggerName;
    }
}
```

The logger that the filter will ignore is specified through the `loggerName` property.

Next you can easily configure the custom filter:

```
<turboFilter class="com.stackify.guide.logging.IgnoreLoggerFilter">
  <LoggerName>colorLogger</LoggerName>
</turboFilter>
```



HTTP REQUEST/RESPONSE LOGGING

Often, you would like to view the details of HTTP requests and responses for your web application. You might consider using custom Logback filters, servlet filters, Spring interceptors, or AOP pointcuts to include this in your logs, but there is an easier way with Logbook.

```
<properties>
  <logbook.version><logbook.version>
</properties>

<dependency>
  <groupId>org.zalando</groupId>
  <artifactId>logbook-core</artifactId>
  <version>${logbook.version}</version>
</dependency>
<dependency>
  <groupId>org.zalando</groupId>
  <artifactId>logbook-servlet</artifactId>
  <version>${logbook.version}</version>
</dependency>
<dependency>
  <groupId>org.zalando</groupId>
  <artifactId>logbook-httpclient</artifactId>
  <version>${logbook.version}</version>
</dependency>
<dependency>
  <groupId>org.zalando</groupId>
  <artifactId>logbook-spring-boot-starter</artifactId>
  <version>${logbook.version}</version>
</dependency>
```

Now, add the following configuration to *logback.xml*. This defines a timestamp (*withDash*) that will be used as a suffix for the logbook file appender that is associated with the logbook default logger category (**org.zalando.logbook.Logbook**) at the TRACE level. Logbook generates a structured JSON object, and we prefix a “msg” key preceded by a “time” key with the timestamp. Then, we wrap the entry in curly braces as a JSON object followed by a comma.

Finally, we can edit the output file to enclose all the records in a JSON array within square brackets, after removing the last comma. You can view these JSON logs with various editors, formatters, and tools; for example, you might try an online [JSON formatter](#) or the [jq command-line JSON processor](#).

```

<timestamp key="withDash" datePattern="yyyy'-'MM'-'dd'-'HH'-'mm'-'ss"/>

<appender name="logbookAppender" class="ch.qos.logback.core.FileAppender">
  <file>logbook-${withDash}.json</file>
  <encoder>
    <pattern>{"time":"%d{HH:mm:ss.SSS}", "msg":%msg},%n</pattern>
  </encoder>
</appender>

<logger level="TRACE" name="org.zalando.logbook.Logbook" additivity="false">
  <appender-ref ref="logbookAppender" />
</logger>

```

After trying *localhost:8888* to get an HTML response, as well as *localhost:8888/greeting* to produce an JSON response, here’s the logbook file as pretty-printed JSON output via jq (with the period as an “all” filter) in the terminal (skipped the HTML headers).

```
jq . logbook-2018-01-23-15-38-48.json
```

You might want use to use pipes for larger files like

```
cat logbook-2018-01-23-15-38-48.txt | jq '.' | less
```

Also Python has a similar built-in JSON pretty printing tool

```
cat logbook-2018-01-23-15-38-48.txt | python -mjson.tool | less
```

Here’s what it looks like.

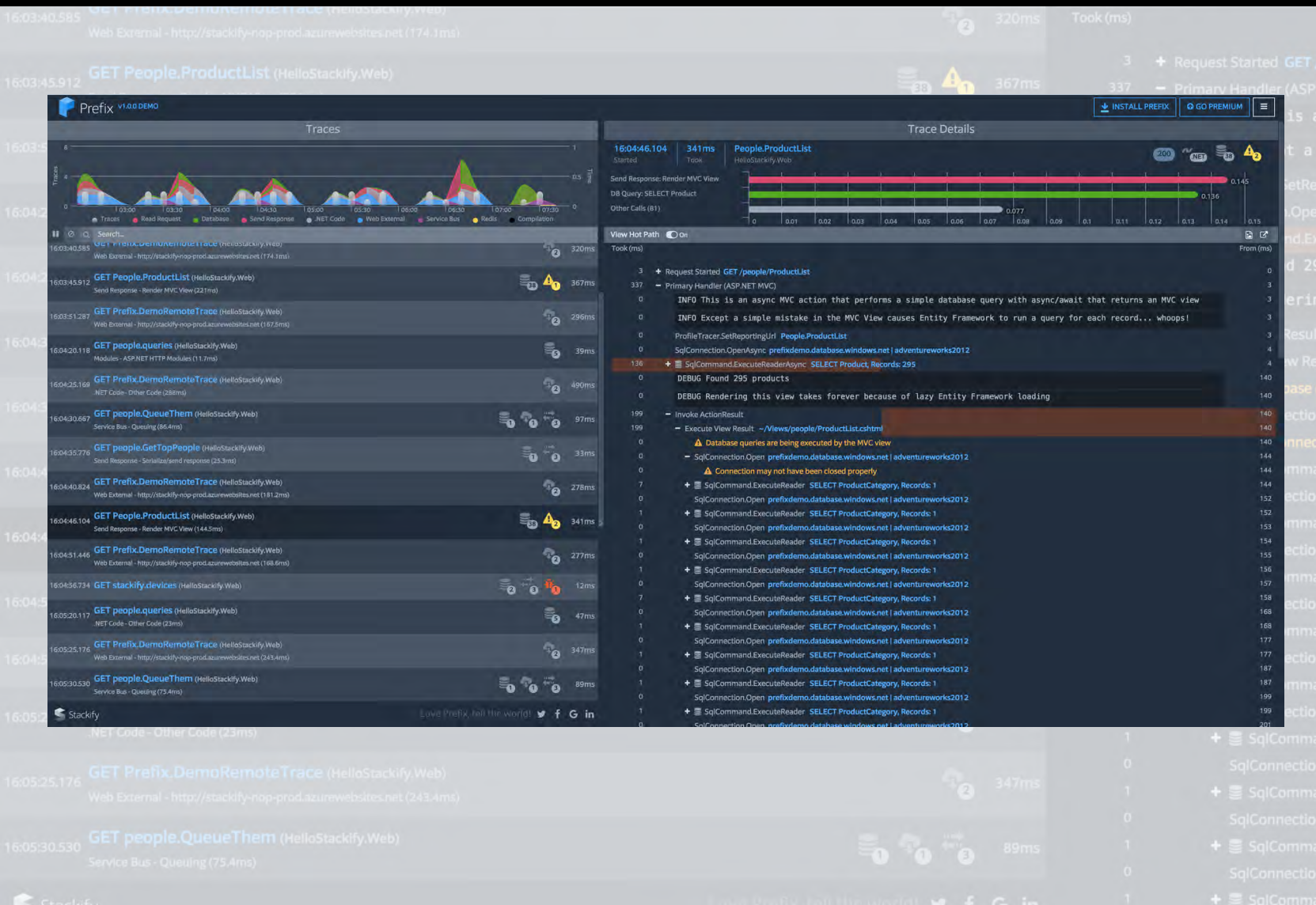
```

[
  {
    "time": "15:39:04.805",
    "msg": {
      "origin": "remote",
      "type": "request",
      "correlation": "184892ce-cc0a-4f9c-91af-8a340045477e",
      "protocol": "HTTP/1.1",
      "remote": "0:0:0:0:0:0:0:1",
      "method": "GET",
      "uri": "http://localhost:8888/",
      "headers": {
        ...
      }
    }
  },

```

```
{
  "time": "15:39:05.001",
  "msg": {
    "origin": "local",
    "type": "response",
    "correlation": "184892ce-cc0a-4f9c-91af-8a340045477e",
    "duration": 222,
    "protocol": "HTTP/1.1",
    "status": 200,
    "headers": {
      "Content-Length": [
        "13"
      ],
      "Content-Type": [
        "text/html; charset=UTF-8"
      ],
      "Date": [
        "Tue, 23 Jan 2018 21:39:04 GMT"
      ],
      "X-Application-Context": [
        "application:8888"
      ]
    },
    "body": "Hello Spring!"
  }
},
{
  "time": "15:39:10.024",
  "msg": {
    "origin": "remote",
    "type": "request",
    "correlation": "d4f67c65-23fa-4d43-9ea8-3800cd797f08",
    "protocol": "HTTP/1.1",
    "remote": "0:0:0:0:0:0:0:1",
    "method": "GET",
    "uri": "http://localhost:8888/greeting",
    "headers": {
      ...
    }
  }
},
```

```
{
  "time": "15:39:10.054",
  "msg": {
    "origin": "local",
    "type": "response",
    "correlation": "d4f67c65-23fa-4d43-9ea8-3800cd797f08",
    "duration": 23,
    "protocol": "HTTP/1.1",
    "status": 200,
    "headers": {
      "Content-Type": [
        "application/json;charset=UTF-8"
      ],
      "Date": [
        "Tue, 23 Jan 2018 21:39:10 GMT"
      ],
      "Transfer-Encoding": [
        "chunked"
      ],
      "X-Application-Context": [
        "application:8888"
      ]
    },
    "body": {
      "id": 1,
      "content": "Hello, Spring!"
    }
  }
}
```



Start seeing what your code is hiding from you.

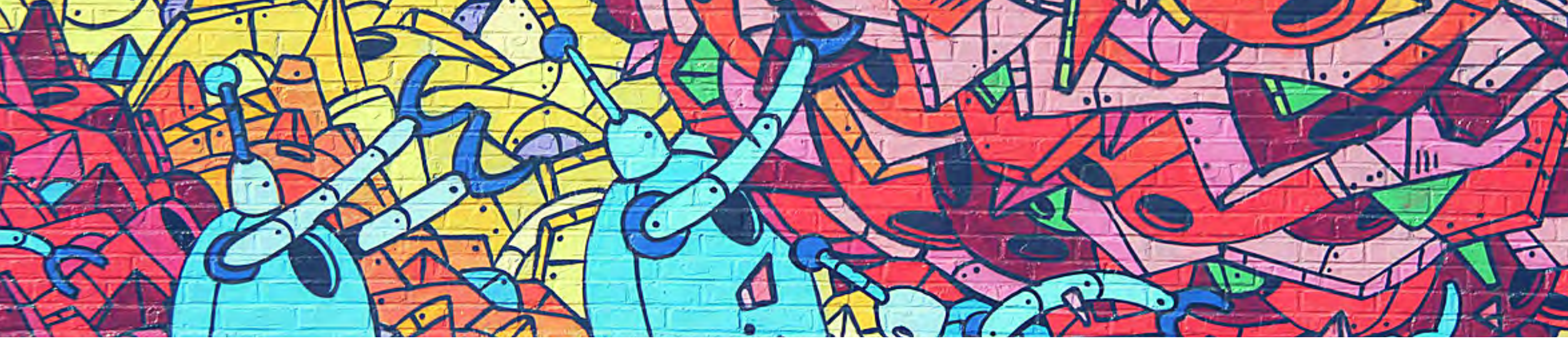
And do it for **FREE**.

[Download now!](#)

"Prefix quickly highlighted that there were hundreds of database calls for a single web method call, something that didn't show up in the standard performance analysis tools.

[A] simple fix literally cut the web method call time in half making me look like an instant hero!"

- Werner van Deventer, DevEnterprise Software



OPERATIONS

Our focus now shifts to operations after applications have been developed, tested, and deployed. It is extremely important to consolidate logs with a centralized cloud platform to effectively manage, monitor, and support application operations in a production environment that supports any business or organization. Stackify Retrace offers a solution for log management and application monitoring that makes this possible. Now let's discuss how that can be done.

GET SERIOUS ABOUT LOGGING

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

Stackify built a "culture of logging" that set out to accomplish these goals:

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

We'll explore these best practices, and share what has been done to enable them, much of which has become part of the Stackify [log management](#) product. Also if you haven't used [Prefix to view your logs](#), be sure to check it out!

LOG EVERYTHING

Start Logging All the Things!

In a lot of shops log messages look like this:

```
} catch (Exception e) {  
    log.error(e.getMessage());  
}
```

Giving the developer credit, at least they are using a try/catch and handling the exception. The exception will likely have a stack trace so you know roughly where it came from, but no other context is logged.

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

```
public void processResults(final List<Double results>) {  
    log.debug("Processing results");  
}
```

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

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

Walk the Code

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

```
public class Foo {  
    private int id;  
    private double value;  
  
    public Foo(int id, double value) {  
        this.id = id;  
        this.value = value;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public double getValue() {  
        return value;  
    }  
}
```

Take the following [factory method](#), which creates a Foo. Note how the door has been opened for error – the method takes a Double as an input parameter. However, when its double value is accessed, there is no check for null and this could cause an exception.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FooFactory {
    public static Foo createFoo(int id, Double value) {
        return new Foo(id, value);
    }
}
```

Note: the original example used `value.doubleValue()` in the return statement. However, it is no longer necessary to explicitly perform this “unboxing” of the double primitive value inside the Double wrapper object. But you can still get a `NullPointerException` from the implicit conversion.

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

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class FooFactory {
    private static Logger log = LoggerFactory.getLogger(FooFactory.class);

    public static Foo createFoo(int id, double value) {

        log.debug("Creating a Foo");
        try {
            Foo foo = new Foo(id, value);
            log.debug("{} ", foo);
            return foo;
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        }
        return null;
    }
}
```

Now, let’s create two Foo’s; one that is valid and one that is not:

```
FooFactory.createFoo(1, Double.valueOf(33.0));
FooFactory.createFoo(2, null);
```

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

```
2017-02-15 17:01:04,842 [main] DEBUG com.stackify.logging.FooFactory: Creating
a Foo
2017-02-15 17:01:04,848 [main] DEBUG com.stackify.logging.FooFactory:
com.stackifytest.logging.Foo@5d22bbb7
2017-02-15 17:01:04,849 [main] DEBUG com.stackify.logging.FooFactory: Creating
a Foo
2017-02-15 17:01:04,851 [main] ERROR com.stackify.logging.FooFactory:
java.lang.NullPointerException
    at com.stackify.logging.FooFactory.createFoo(FooFactory.java:15)
    at com.stackify.logging.FooFactoryTest.test(FooFactoryTest.java:11)
```

Now we have some logging – we know when Foo objects are created, and when they fail to be created in createFoo(). But we are missing some context that would help. The default toString() implementation doesn’t build any data about the members of the object, but the class name and its unique address joined with an “at” symbol (@), which isn’t very useful. We have some options here, but let’s have the IDE generate an implementation for us.

In Eclipse, right-click the class name and select “Source > Generate toString()...” where there are several options such string concatenation (like here) or StringBuilder for more complex objects; in IntelliJ IDEA, select Code (or right click) > Generate... > toString(). Notice that Eclipse creates Foo [id=1, value=33.0] with square brackets (slightly misleading since it’s not an array), whereas IDEA produces “Foo {id=1, value=33.0}” with curly braces (more appropriate for an object, similar to JSON format). Similar additional options are available to generate standard constructor, getters, setters, hashCode, and equals methods.

```
@Override
public String toString() {
    return "Foo [id=" + id + ", value=" + value + "]";
}
```

Run our test again:

```
2017-02-15 17:13:06,032 [main] DEBUG com.stackify.logging.FooFactory: Creating
a Foo
2017-02-15 17:13:06,041 [main] DEBUG com.stackify.logging.FooFactory: Foo
[id=1, value=33.0]
```

```
2017-02-15 17:13:06,041 [main] DEBUG com.stackify.logging.FooFactory: Creating
a Foo
2017-02-15 17:13:06,043 [main] ERROR com.stackify.logging.FooFactory:
java.lang.NullPointerException
    at com.stackify.logging.FooFactory.createFoo(FooFactory.java:15)
    at com.stackify.logging.FooFactoryTest.test(FooFactoryTest.java:11)
```

Much better! Now we can see the object that was logged as “[id=, value=]”. Another option you have for toString is to use Java’s reflection capabilities. The main benefit here is that you don’t have to modify the toString method when you add or remove members.

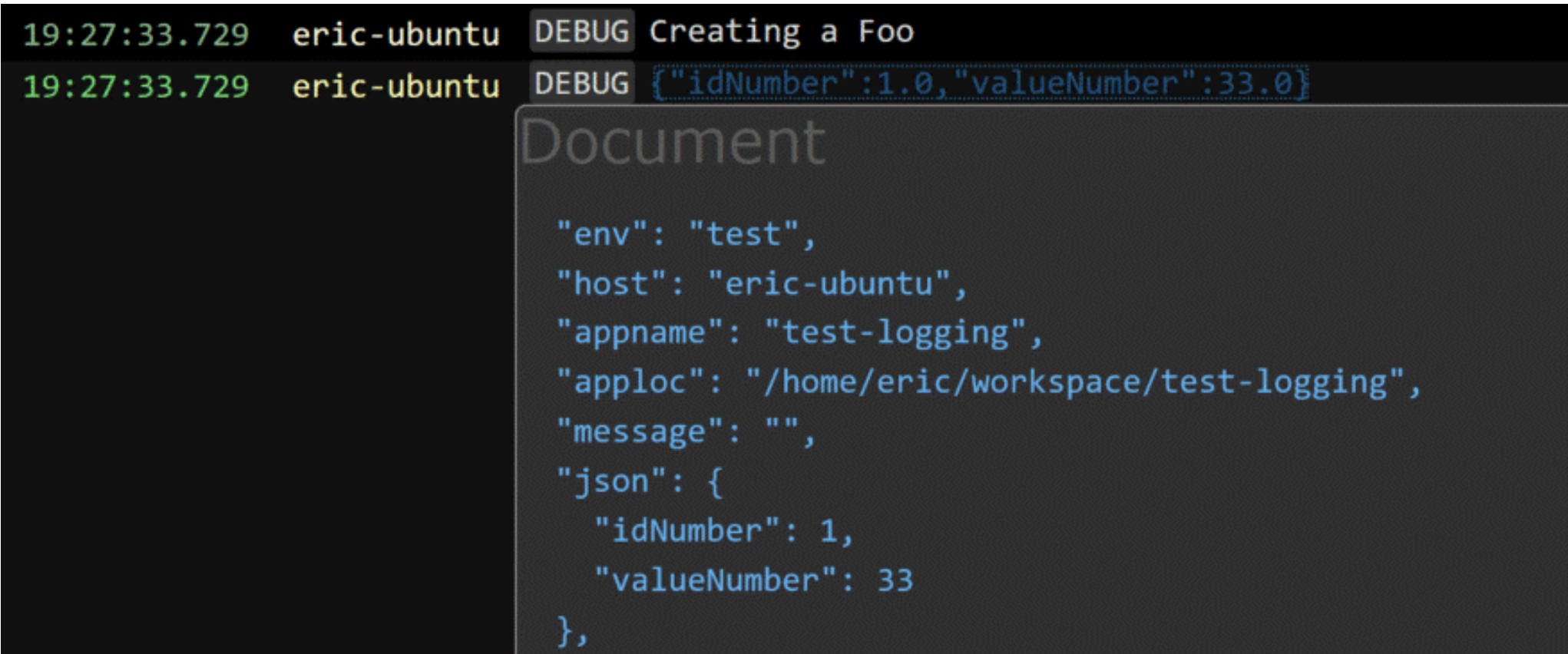
Here is an example using Google’s Gson library which converts Java objects between JSON strings. Changing toString in Foo produces JSON.

```
public String toString() {
    return gson.toJson(this);
}
```

Now, let’s look at the output:

```
2017-02-15 17:22:55,584 [main] DEBUG com.stackifytest.logging.FooFactory:
Creating a Foo
2017-02-15 17:22:55,751 [main] DEBUG com.stackifytest.logging.FooFactory:
{"id":1,"value":33.0}
2017-02-15 17:22:55,754 [main] DEBUG com.stackifytest.logging.FooFactory:
Creating a Foo
2017-02-15 17:22:55,760 [main] ERROR com.stackifytest.logging.FooFactory:
java.lang.NullPointerException
    at com.stackifytest.logging.FooFactory.createFoo(FooFactory.java:15)
    at com.stackifytest.logging.FooFactoryTest.test(FooFactoryTest.java:11)
```

When you log objects as JSON and use Stackify’s Retrace tool, you can get some nice details like this.



Retrace Logging Dashboard JSON Viewer

Diagnostic Contexts

And this brings us to one last point on logging more details: diagnostic context logging. When it comes to debugging a production issue, you might have the “Creating a Foo” message thousands of times in your logs, but with no clue who the logged-in user was that created it. Knowing the user is priceless context to quickly resolve an issue. Think about what other detail might be useful like `HttpRequest` details. But who wants to remember to log it every time? Diagnostic context logging comes to the rescue, specifically the mapped diagnostic context. Read more about SLF4J’s MDC here: <https://logback.qos.ch/manual/mdc.html>.

The easiest way to add context items to your logging for web applications is usually a servlet filter. For this example, let’s create a servlet filter that generates a transaction id and attaches it to the MDC.

```
public class LogContextFilter implements Filter {
    public void init(FilterConfig config) {}
    public void destroy() {}

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws ServletException, IOException {
        String transactionId = UUID.randomUUID().toString();
        MDC.put("TRANS_ID", transactionId);
        try {
            chain.doFilter(request, response);
        } finally {
            MDC.clear();
        }
    }
}
```

Now, we can see some log statements like this:

```
19:49:36.350 eric-ubuntu DEBUG Creating a Foo {"TRANS_ID":"d02c0890-b661-4460-7a0c5b3d1763"}
19:49:36.406 eric-ubuntu DEBUG {"id":"1","value":33.0} {"TRANS_ID":"d02c0890-b661-4460-7a0c5b3d1763"}
```

More context. We can now trace all log statements from a single request.

Before moving to the next topic, let’s address a question that you might be asking: “But if I log everything, won’t that create overhead, unnecessary chatter, and huge log files?” Our answer comes in a couple of parts: first, use the logging verbosity levels. You can `log.debug()` everything that you think

you'll need, and then set your configuration for production appropriately, i.e. Warning and above only. When you do need the debug info, it's only changing a config file and not redeploying code. Second, if you're logging in an *async, non-blocking* way, then overhead should be low. Last, if you're worried about space and log file rotation, there are smarter ways to do it, and we'll talk about that in the next section.

APPLICATION SUPPORT

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

- There's a lot more data.
- You have to get access to that data.
- It's spread across multiple servers.
- A specific operation may be spread across applications – so there are even more logs to dig through.
- It's flat and hard to query; even if you do put it in a database, you are going to have to do full-text indexing to make it usable.
- It's hard to read; messages are scrambled.
- You generally don't have any context of the user, etc.
- You probably lack some details that would be helpful. You mean `log.info("In the method")` isn't helpful???
- You will be managing log file rotation and retention.

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

WORK SMARTER

Now that we're logging everything, and it's providing more contextual data, we're going to look at the next part of the equation. As we've mentioned and demonstrated, just dumping all of this out to flat files still doesn't help you out a lot in a large, complex application and environment.

Factor in thousands of requests, files spanning multiple days, weeks, or longer, and across multiple servers, and you have to consider how you are going to quickly find the data that you need.

What we all really need is a solution that

- Aggregates all log & exception data to one place
- Makes it available instantly to everyone on your team
- Presents a timeline of logging throughout your entire stack/infrastructure
- Is highly indexed and searchable by being in a structured format

STACKIFY RETRACE

This is where we tell you about [Stackify Retrace](#). As we sought to improve our own abilities to quickly and efficiently work with our log data, we decided to make it a core part of our product; yes, we use Stackify to monitor Stackify! We share this with our customers since we believe it's an issue central to application troubleshooting.

Custom Logging Appenders

First, we realize that lots of developers already have logging in place, and aren't going to want to take a lot of time to rip that code out and put new code in. That's why we've created logging appenders for the most common Java logging frameworks.

- logback (<https://github.com/stackify/stackify-log-logback>)
- log4j 2.x (<https://github.com/stackify/stackify-log-log4j2>)
- log4j 1.2 (<https://github.com/stackify/stackify-log-log4j12>)

Direct Log4J12 Appender

Continuing with log4j as a sample, the setup is easy. Just add the Stackify appender to your project's Maven POM file.

```
<dependency>
  <groupId>com.stackify</groupId>
  <artifactId>stackify-log-log4j12</artifactId>
  <version>2.0.2</version>
  <scope>runtime</scope>
</dependency>
```

Also, add in some Log4J configuration for the Stackify appender to your logging.properties file.

```
log4j.rootLogger=DEBUG, CONSOLE, STACKIFY
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d [%t] %-5p %c: %m%n
log4j.appender.STACKIFY=com.stackify.log.log4j12.StackifyLogAppender
log4j.appender.STACKIFY.apiKey=[HIDDEN]
log4j.appender.STACKIFY.environment=test
```

Direct Logback Appender

Stackify also provides an appender for Logback, which is included in the Maven POM file:

```
<dependency>
  <groupId>com.stackify</groupId>
  <artifactId>stackify-log-logback</artifactId>
  <version>2.0.2</version>
  <scope>runtime</scope>
</dependency>
```

Here is a sample XML configuration of the Logback appender.

```
<appender name="STACKIFY"
class="com.stackify.log.logback.StackifyLogAppender">
  <apiKey>YOUR_ACTIVATION_KEY</apiKey>
  <application>YOUR_APPLICATION_NAME</application>
  <environment>YOUR_ENVIRONMENT</environment>
</appender>
...
<root level=...>
  ...
  <appender-ref="STACKIFY">
</root>
```

As you can see, if you’re already using a different appender, you can keep it in place and put them side-by-side.

Direct Log4J2 Appender

Stackify also provides an appender for Log4J2, with this dependency in the Maven POM file:

```
<dependency>
  <groupId>com.stackify</groupId>
  <artifactId>stackify-log-log4j2</artifactId>
  <version>2.0.2</version>
  <scope>runtime</scope>
</dependency>
```

Here is a sample XML configuration of the Log4J2 appender.

```
<Configuration packages="com.stackify.log.log4j">
  <Appenders>
    <StackifyLog name="STACKIFY"
      apiKey="YOUR_API_KEY"
      application="YOUR_APPLICATION_NAME"
      environment="YOUR_ENVIRONMENT"/>
    ...
  </Appenders>
```

```
<Loggers>
  <Root ...>
    ...
    <AppenderRef ref="STACKIFY"/>
  </Root>
</Loggers>
</Configuration>
```

Data Masking

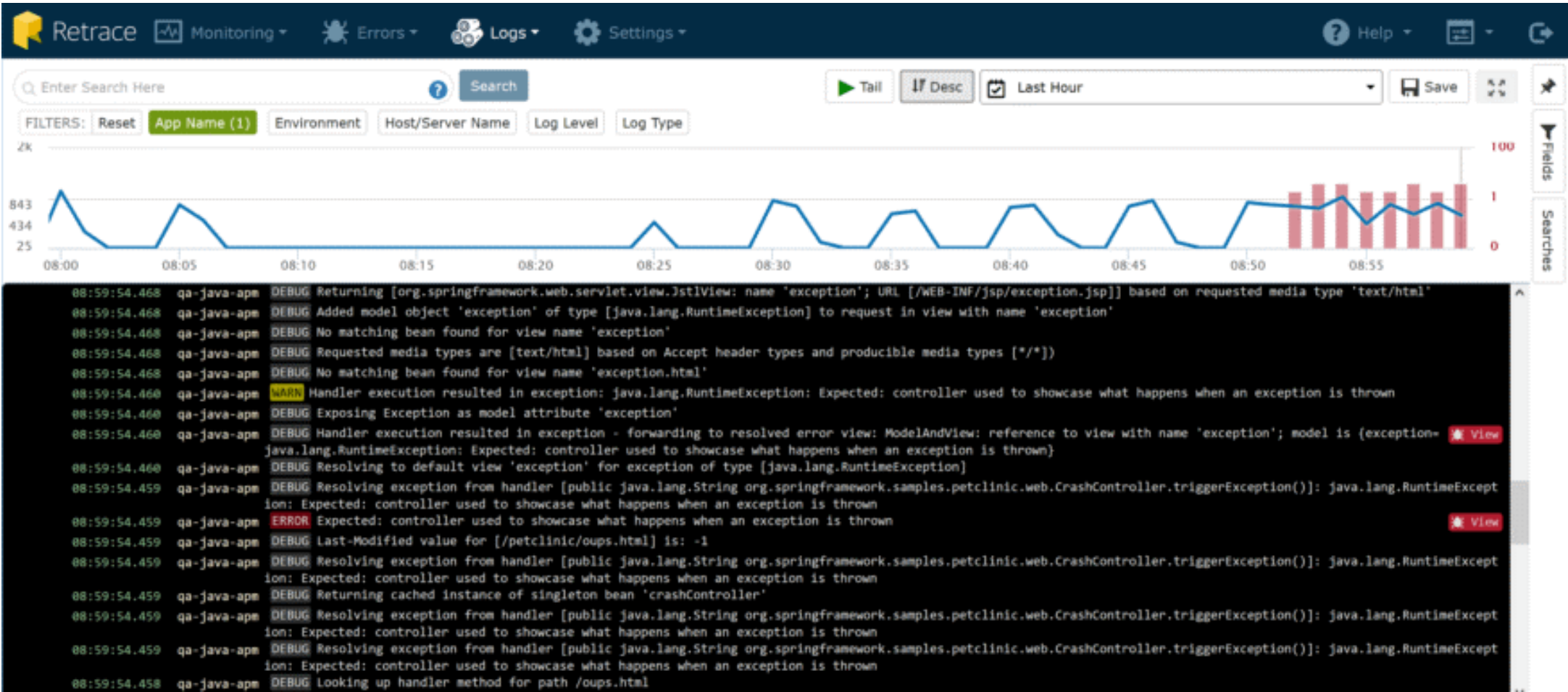
The Stackify appenders for both Log4J and Logback also have built-in data masking of sensitive information, such as credit card and social security numbers, IP addresses, and custom masks using regular expression (regex) matching (like removing vowels).

Standalone Logwatcher

Recently, Stackify also released a [standalone logwatcher](#), which can be useful for apps that you can't inject an appender into (legacy). Its main purpose is to monitor files that you consider important, like Apache web server or system logs (syslog). You can use it to upload the contents of these files as they are updated to Retrace, where the contents will then be viewable via the Logs Dashboard. Only one file can be monitored per path, so if you have multiple files you wish to monitor, then you must configure the path for each of them separately.

Logging Dashboard

Now that you’ve got your logs streaming to Stackify, we can take a look at the logging dashboard. By the way, if our monitoring agent is installed, you can also send [Syslog](#) entries to Stackify as well!



This dashboard shows a consolidated stream of log data, coming from all your servers and apps, presented in a timeline.

If you’re a developer, team lead, or architect, Stackify’s tools were built for you. In fact, we have two game-changing, code performance products no developer or dev team should ever be without.



Prefix

Prefix is a popular developer tool for finding and fixing bugs while you write your code. You have profilers and debuggers, but nothing is like Prefix.

[Download now for FREE.](#)



Retrace

Retrace is an APM tool built specifically for dev teams. Our agent can install on pre-prod or production servers to find and fix problems faster.

[Start your 14-day trial.](#)

LOG MANAGEMENT

Centralized Cloud Solution

Having consolidated logs available at a centralized Stackify cloud server simplifies log management and enables application monitoring.

This enables you to filter and search logs, explore exceptions, and monitor errors, and track metrics.

Filtering Logs

From the dashboard, you can quickly

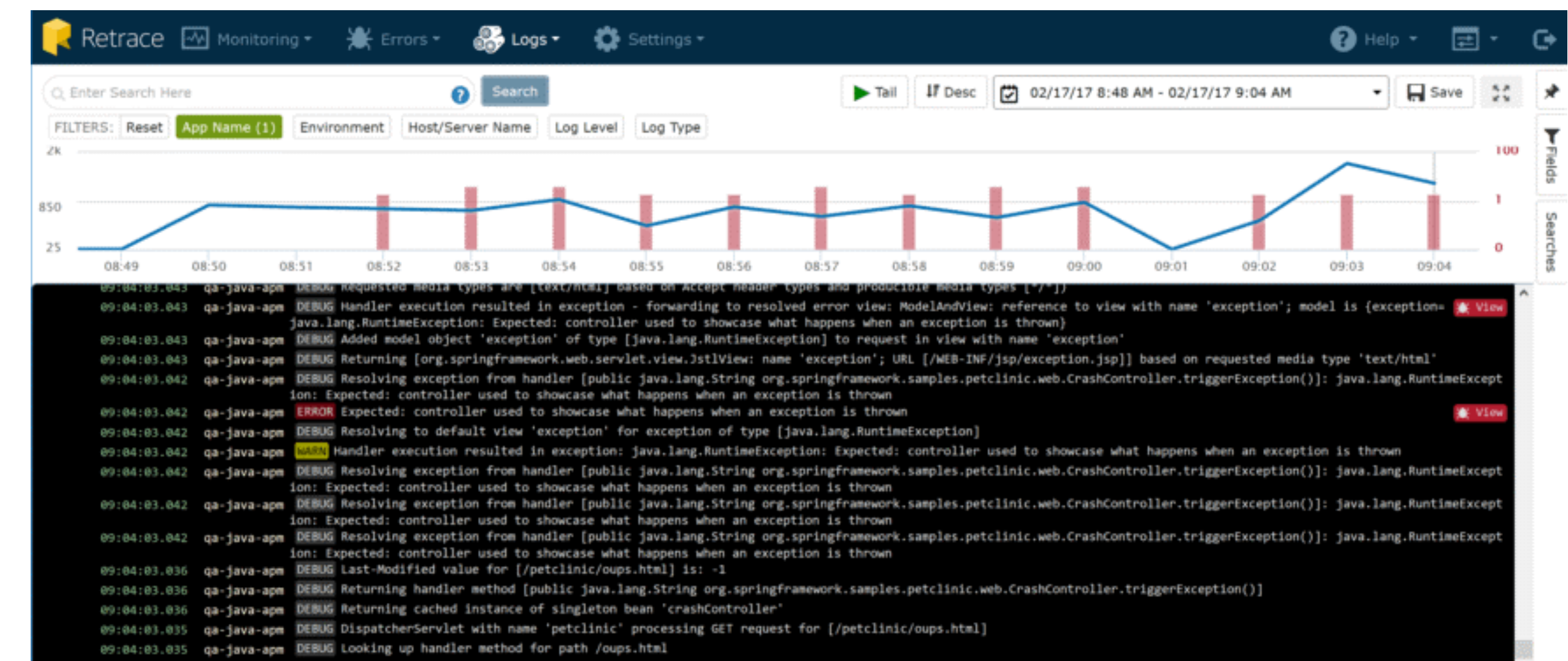
- View logs based on a range of time
- Filter for specific servers, apps, or environments

Plus, there are a couple of really great usability things built in. One of the first things that you’ll notice is the chart at the top. It’s a great way to quickly “triage” your application. The blue line indicates the rate of log messages, and the red bars indicate the number of [exceptions being logged](#).

Examining Exceptions and Time Periods

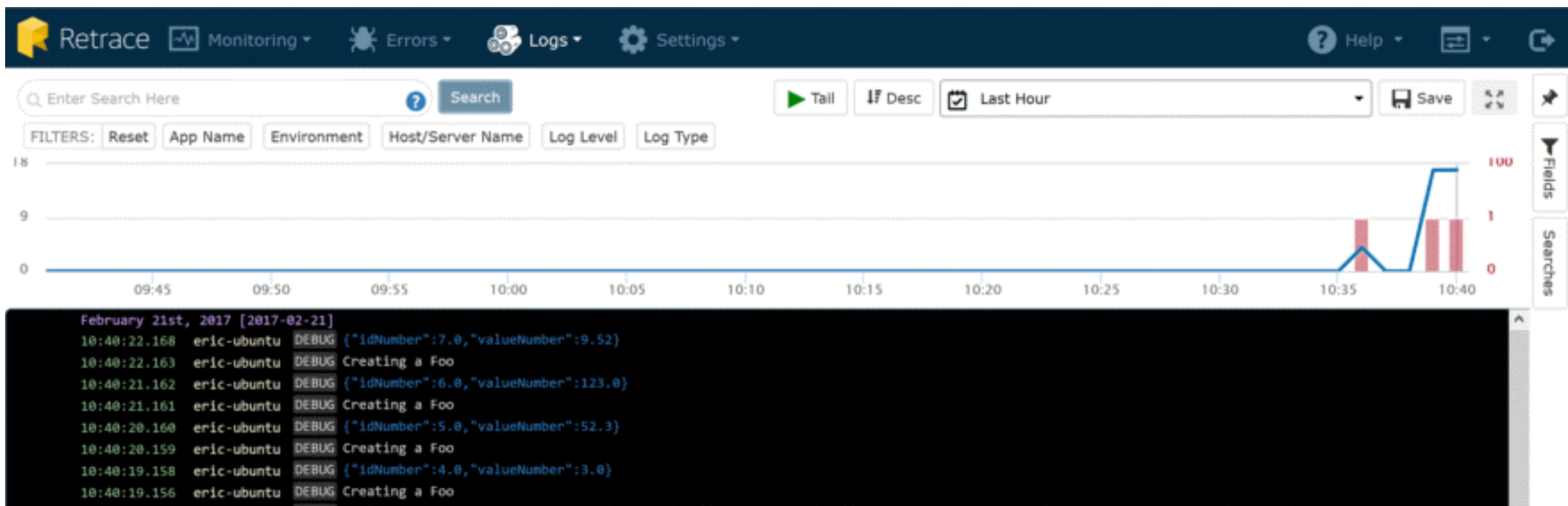
It’s clear that a few minutes ago, the web app started having more consistent activity but more importantly, we started getting more exceptions at the same time. Exceptions don’t come without overhead for your CPU and memory, and they also can have a direct impact on user satisfaction, which can cost real money.

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



Searching Your Logs

Do you see that blue text below that looks like a JSON object?

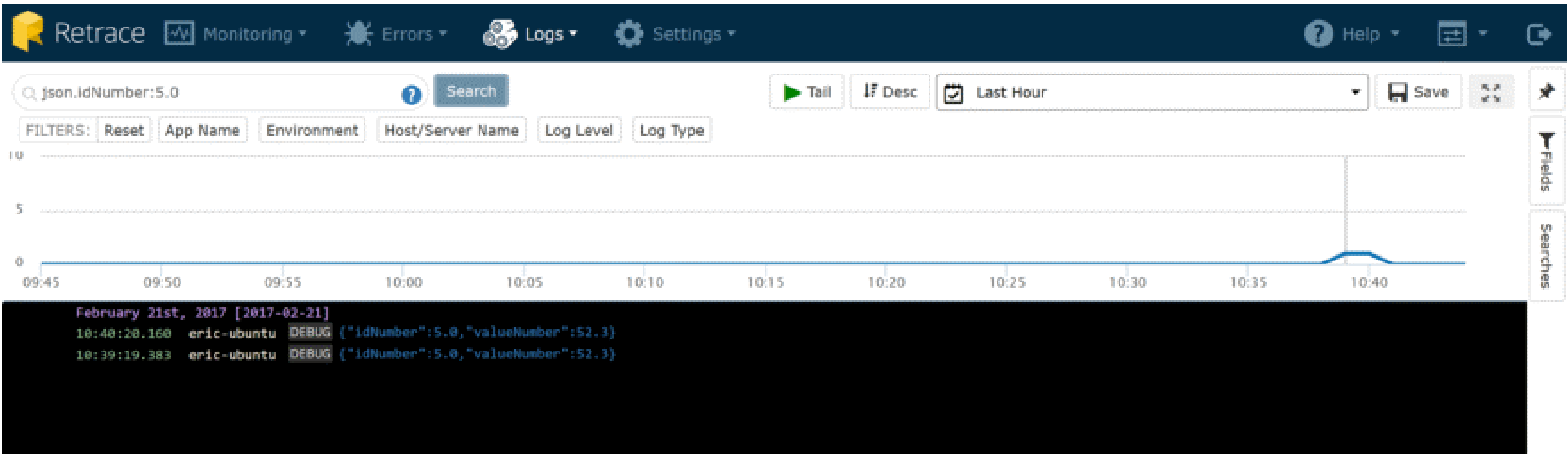


Well, it *is* a JSON object. That’s the result of logging objects, and adding context properties earlier. It looks a lot nicer than plain text in a flat file, doesn’t it? Well, it gets even more awesome. See the search box at the top of the page? You can put in any search string that you can think of, and it will query all your logs as if it were a flat file.

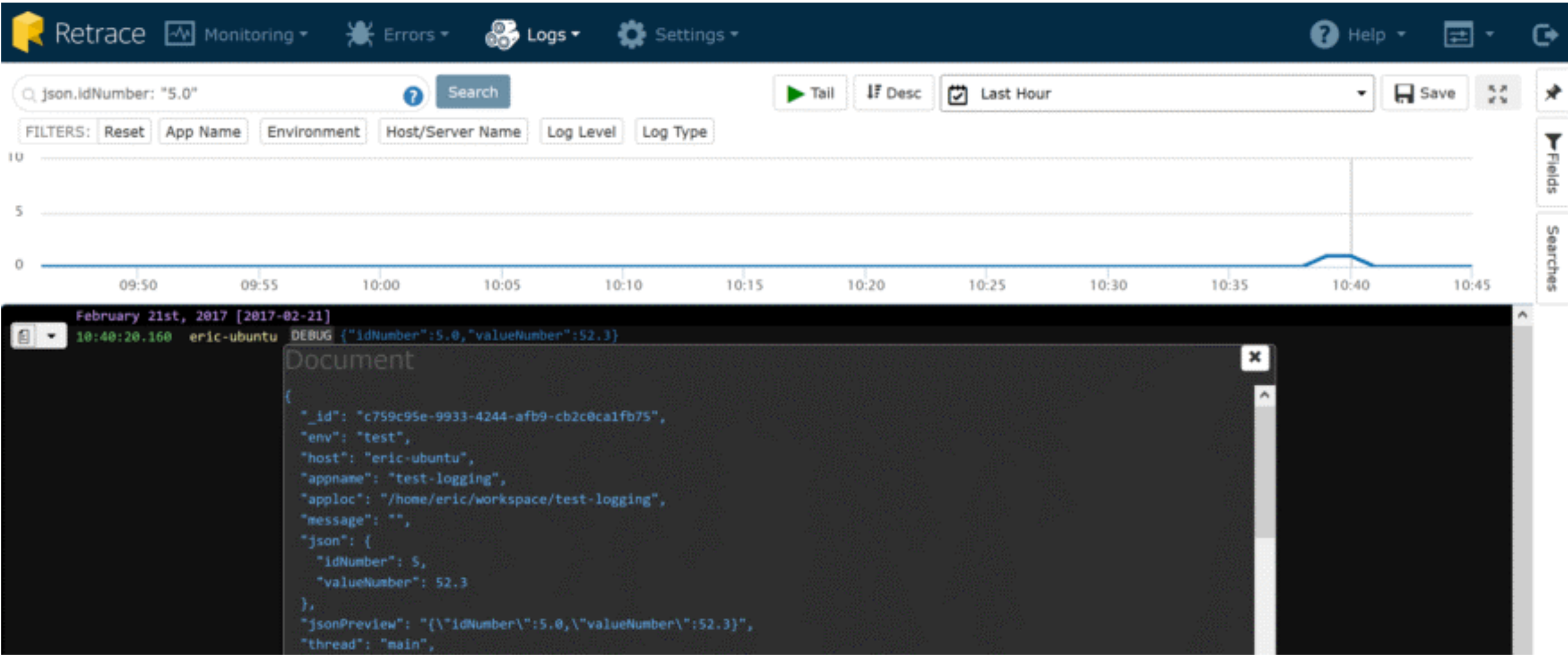
However, as we discussed earlier, this isn’t great because you could end up with a lot more matches than you want. Suppose that you want to search for all objects with an id of 5. Fortunately, our log aggregator is smart enough to help in this situation. That’s because when we find serialized objects in logs, we index each and every field we find. That makes it easy to perform a search like this:

json.idNumber:5.0

That search yields the following results:

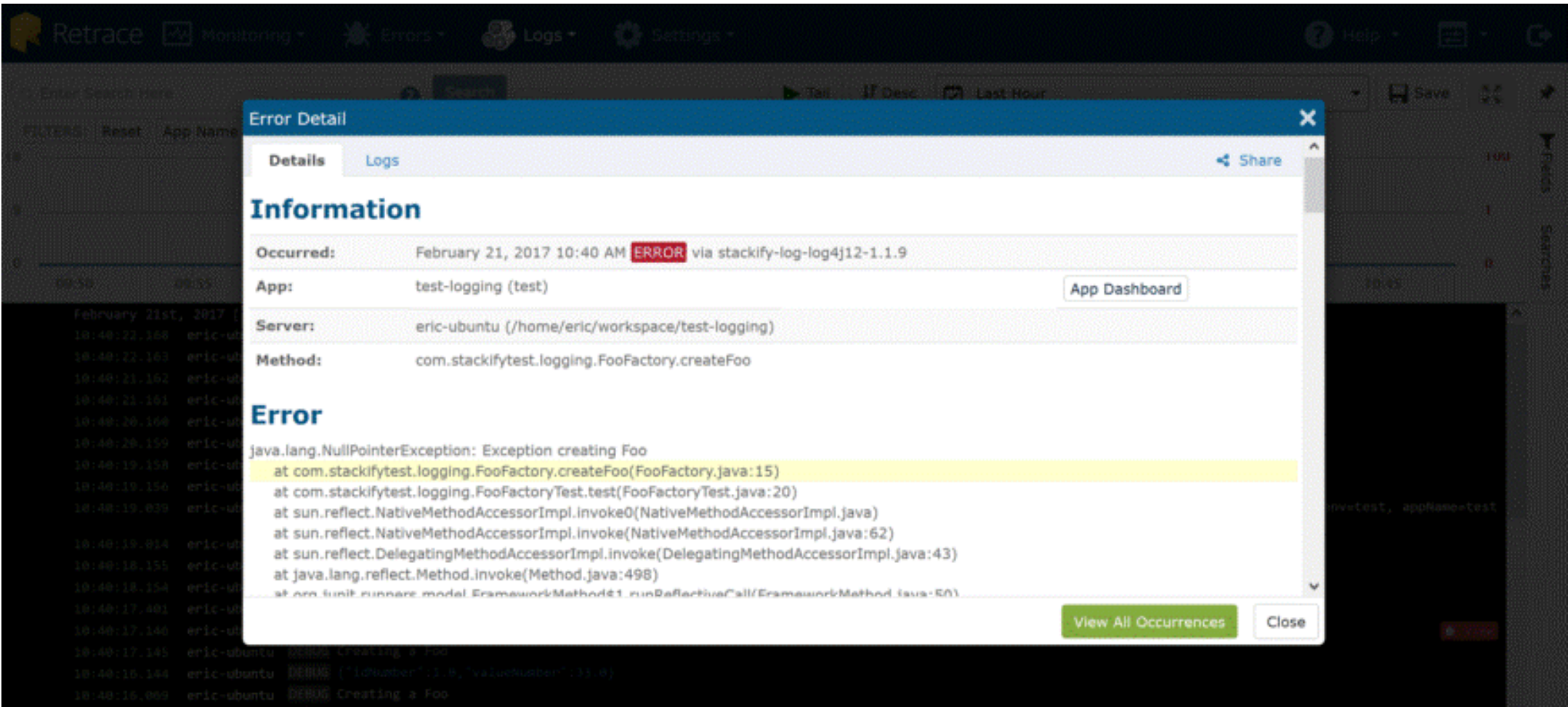


Let's look at what else you can search by. Click on the document icon when you hover over a log record, and you'll see all the fields that Stackify indexes. This allows you to get more value out of your logs and search by all the fields, otherwise known as [structured logging](#).

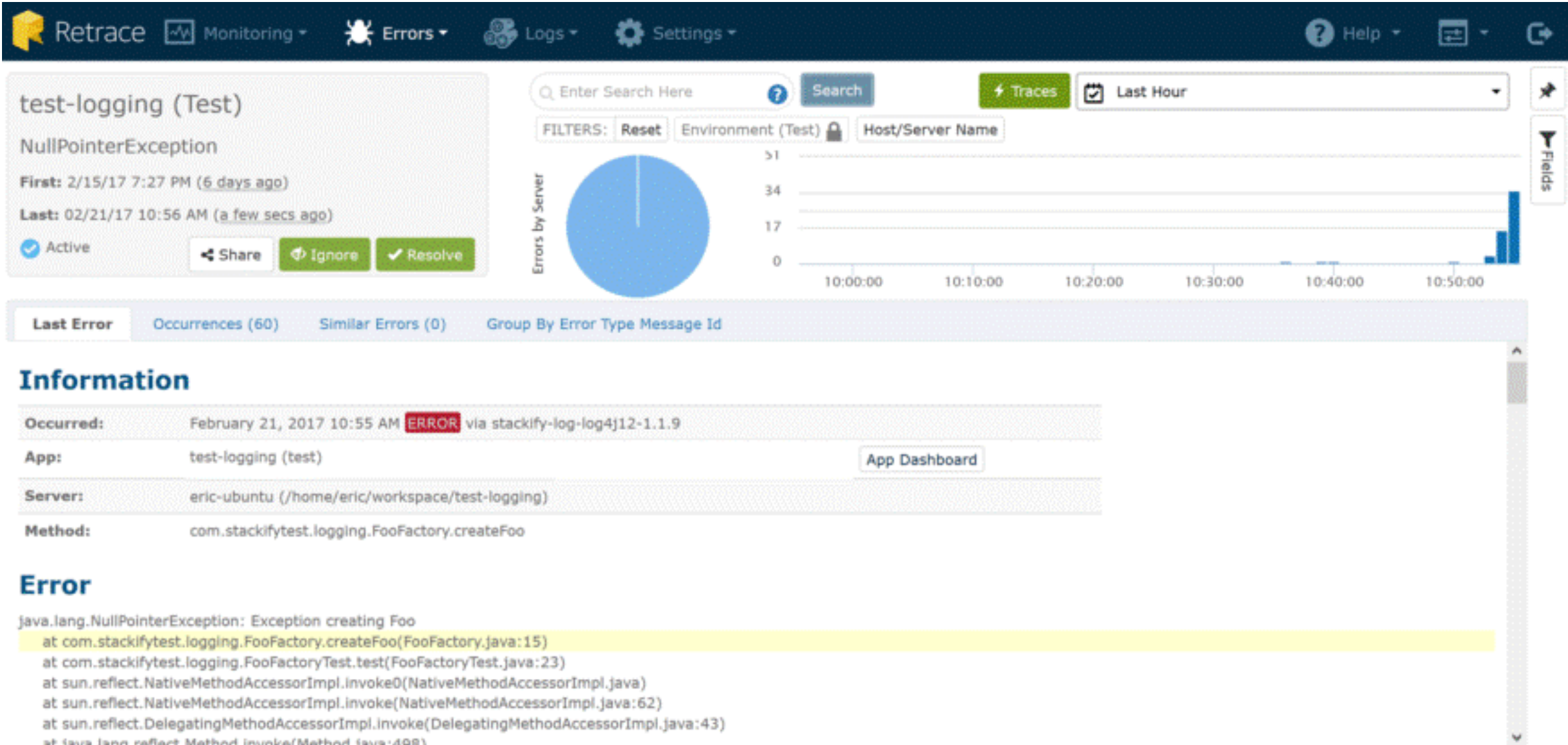


Exploring Java Exception Details

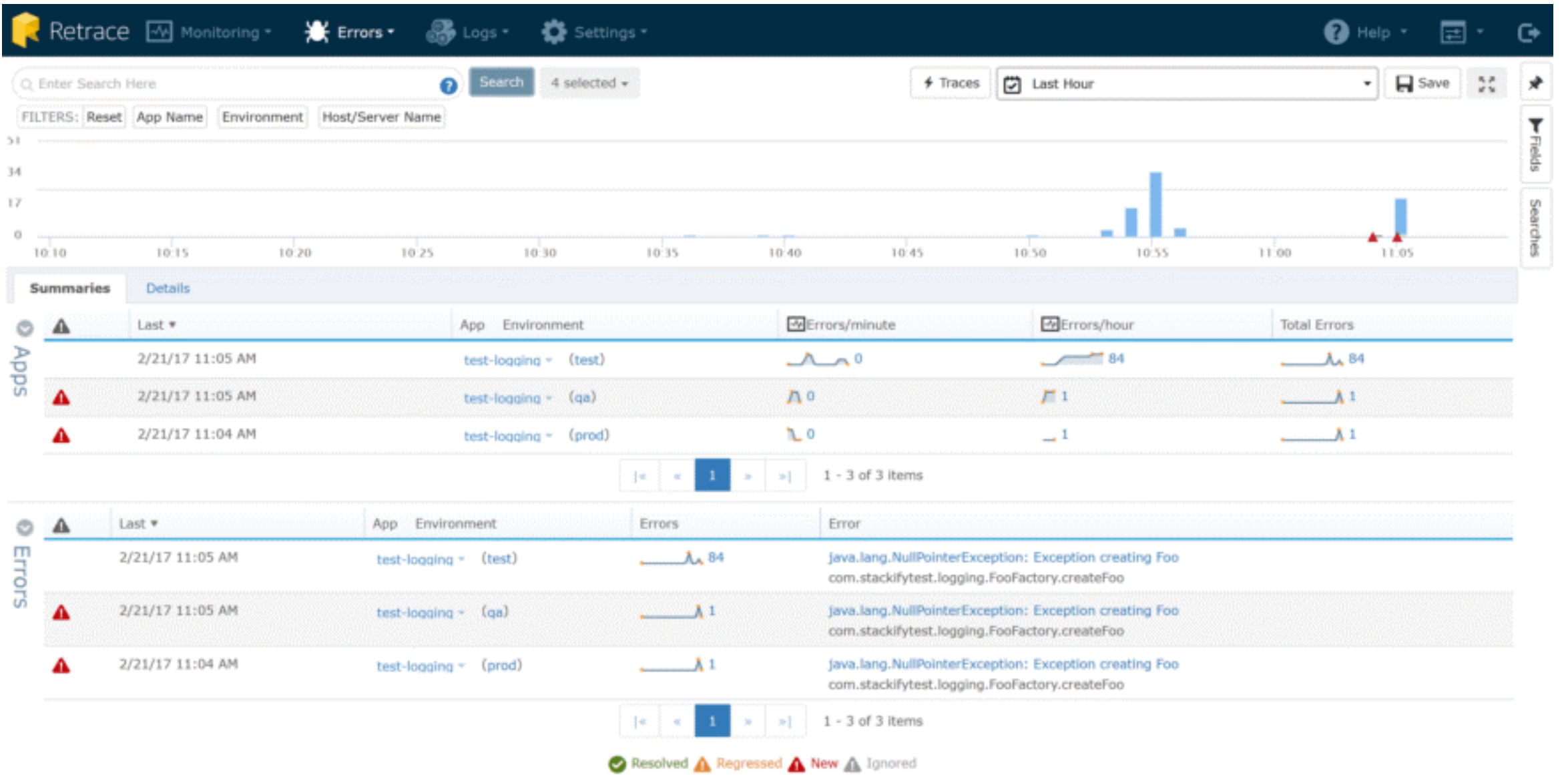
You may have also noticed this little red bug icon (🐛) next to exception messages. That's because we treat exceptions differently by automatically showing more context. Click on it and we present a deeper view of that exception.



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



You can quickly see this error has occurred 60 times over the last hour. Errors and logs are closely related, and in an app where a tremendous amount of logging can occur, exceptions could sometimes get a bit lost in the noise. That’s why we’ve built an [Errors Dashboard](#) as well, to give you this same consolidated view but limited to exceptions.



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

- uptick in the rate of exceptions over the past few minutes
- majority of errors are coming from “test” environment – about 84 per hour
- couple of new errors just started occurring (indicated by red triangles)

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

APPLICATION MONITORING

You can establish monitor thresholds for errors & rates, and receive notification alerts when they are reached.

Monitors & Alerts

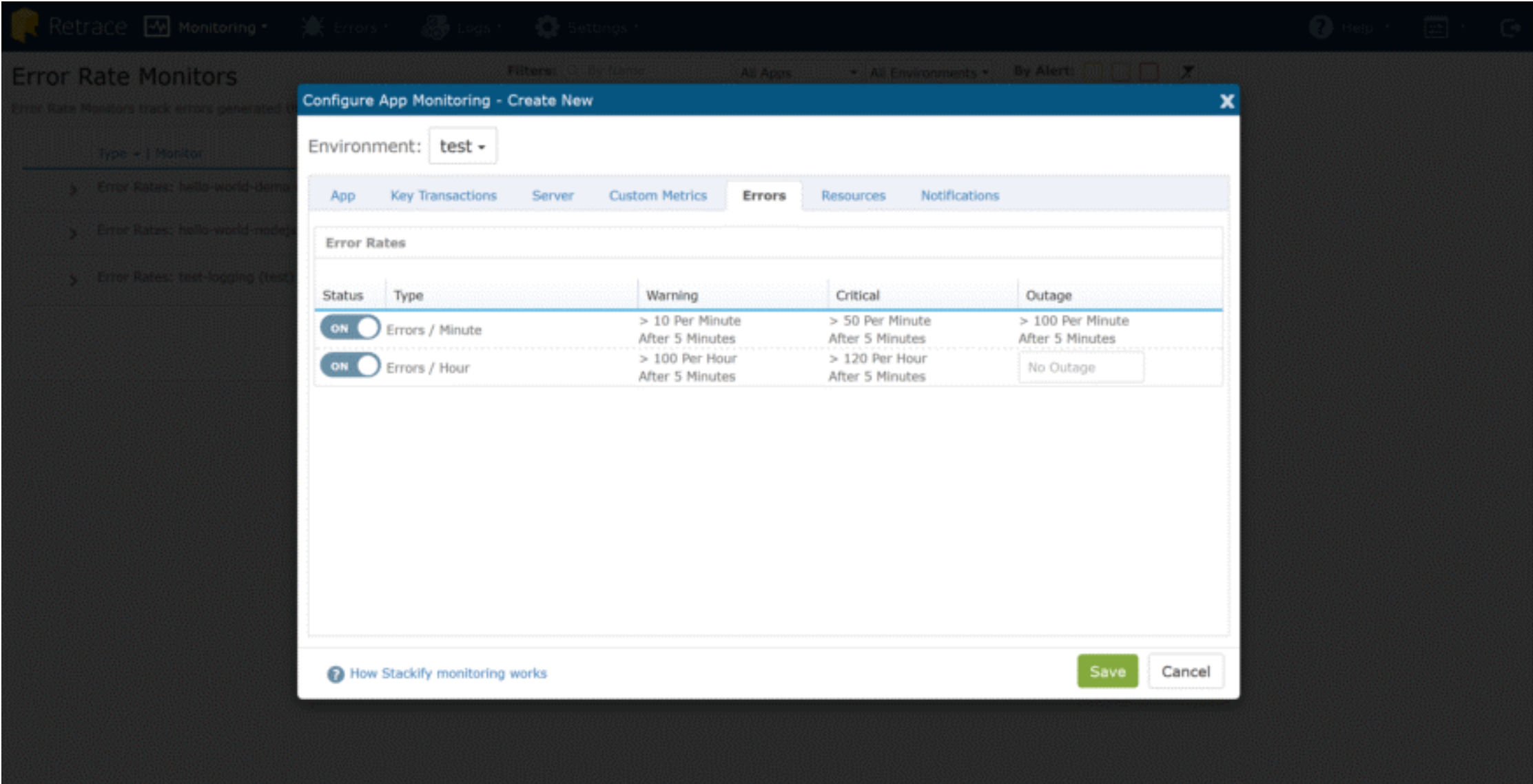
Wouldn’t it be nice to be alerted when

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

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

Error Rates

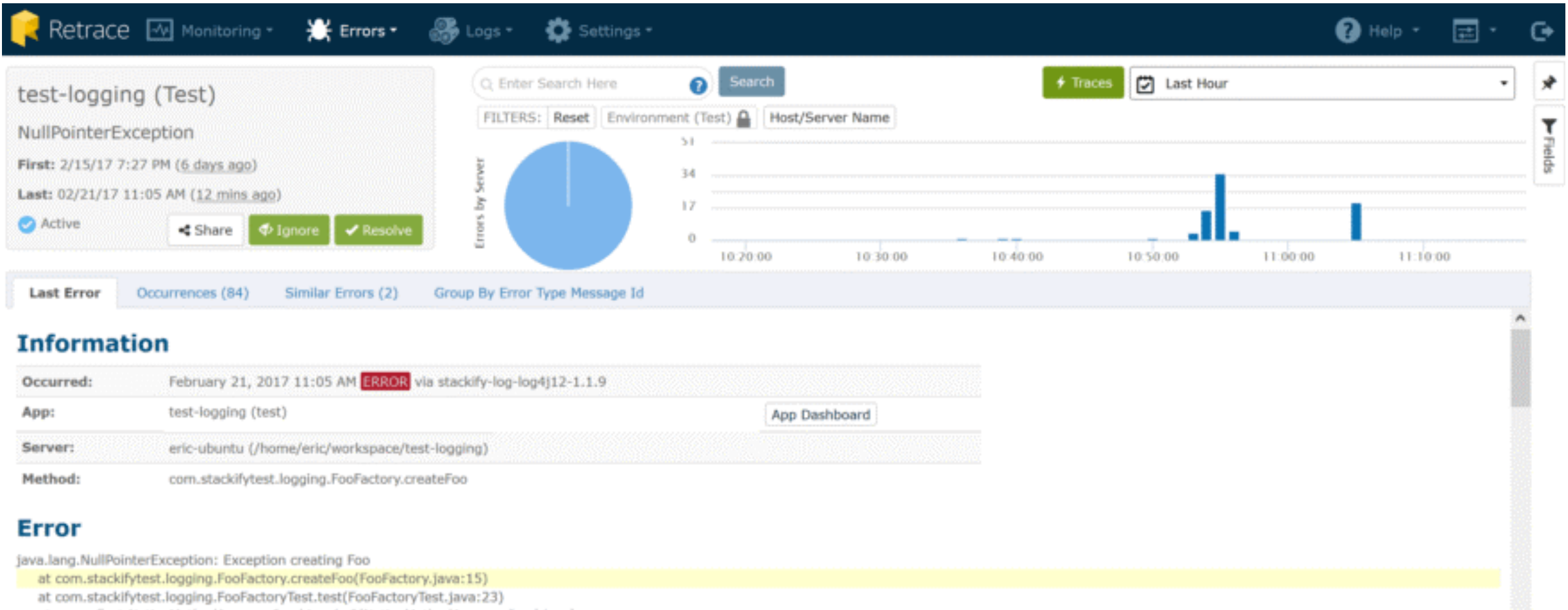
Let's look at the error dashboard, where the ‘test’ environment is getting a high number of errors per hour. From the Error dashboard, click on “Error Rates” and then select which app/environment you wish to configure alerts for.



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

Resolved Errors & New Errors

Earlier on, we introduced a new error by not checking for null values when creating Foo objects. We’ve since fixed that and confirmed it by looking at the details for that particular error. As you can see, the last time it happened was 12 minutes ago:



It was a silly mistake, but one that is easy to make. You can mark this one as “resolved”, which lets you do something really cool: get an alert if it comes back. The Notifications menu will let you check your configuration, and by default, you’re set to receive both new and regressed error notifications for all your apps and environments.

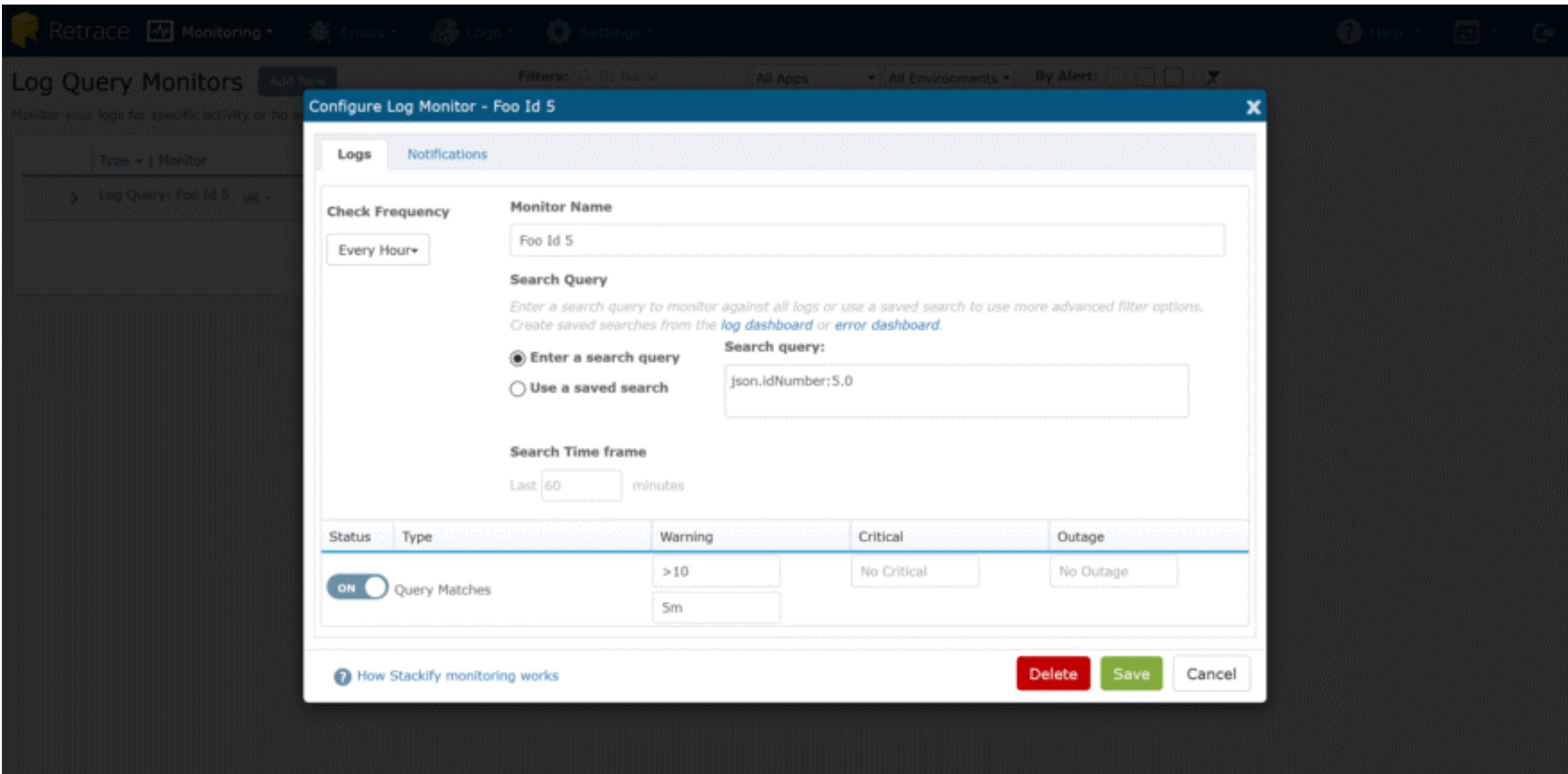
Error Notifications Add New				
For each notification group, configure which environments and applications they should receive emails for when a new alert occurs or if a fixed error regresses.				
Notification Group	Environments	Apps	New Errors	Regressed Errors
Eric	Test	test-logging	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>



Log Monitors

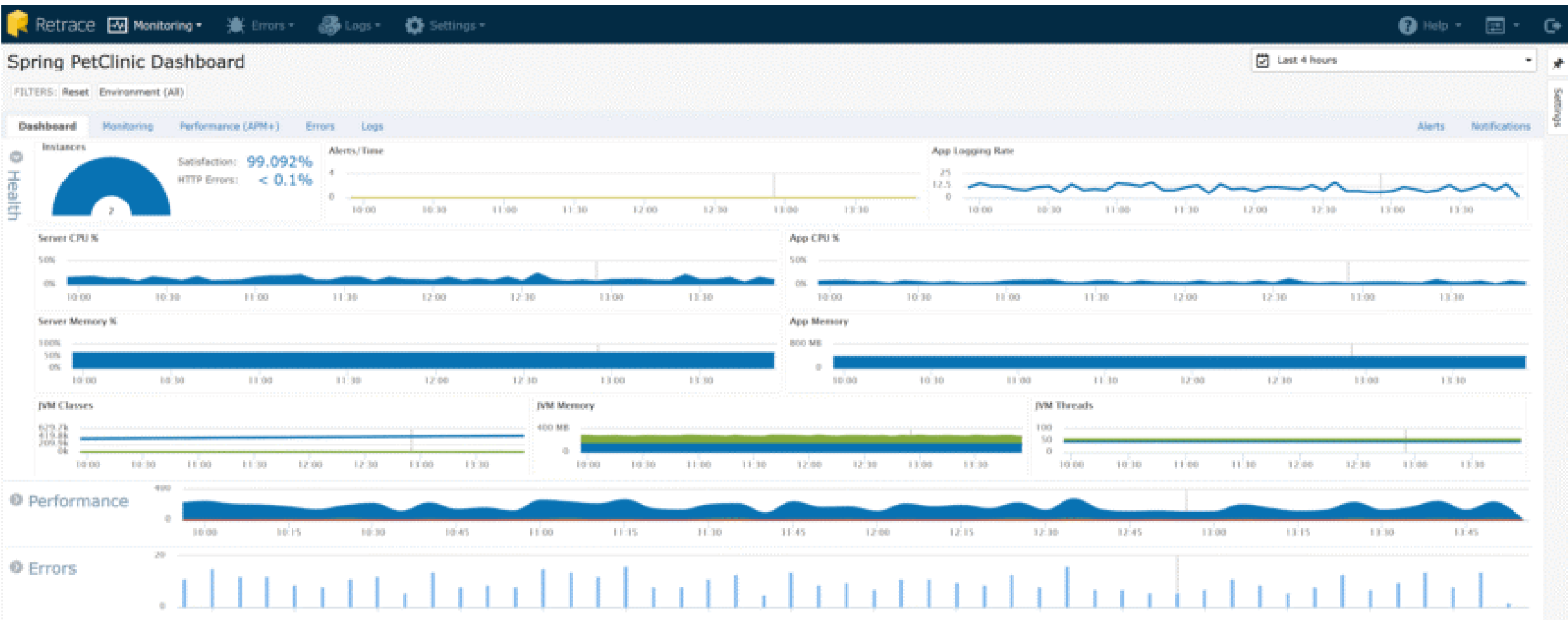
Some things aren't very straightforward to monitor. Perhaps you have a critical process that runs asynchronously and the only record of its success (or failure) is logging statements. Earlier, we discussed the ability to run deep queries against your [structured log data](#), and any of those queries can be saved and monitored. Here's a very simple scenario: the query is executed every minute, and we can monitor how many matching records we have.

It's just a great simple way to check system health if a log file is your only indication.

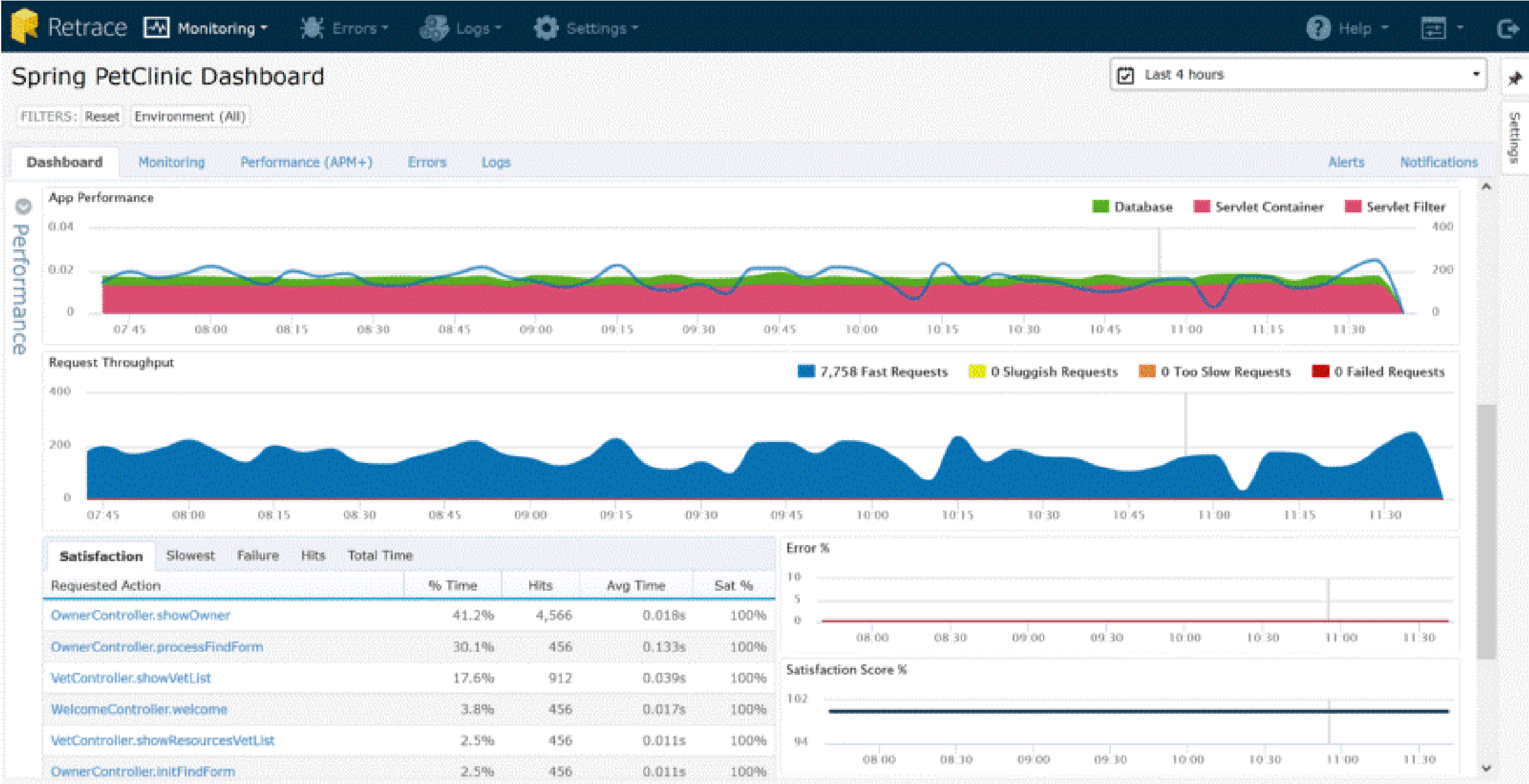


Java Logging Best Practices

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



As you can see, you get some great contextual data at a glance that errors and logs contribute to: Satisfaction and HTTP Error Rate. You can see that user satisfaction is high and the HTTP error rate is low. You can quickly start drilling down to see which pages might not be performing well, and what errors are occurring.





CONCLUSION

There was a lot to cover on logging, and we've barely scratched the surface. If you dig a little deeper or even get your hands on it, you can! We hope that these Java logging best practices will help you write better logs and save time troubleshooting.

All of our Java logging appenders are available on [GitHub](#) and you [can sign up for a free trial](#) to get started with Stackify today!

You have a number of options to choose from when it comes to logging in the Java ecosystem. Out of all of these, Logback is certainly a great choice and a very powerful library. It also brings several improvements over *Log4J*, such as better overall performance, more advanced filtering possibilities, automatic reloading of configuration files, automatic removal of archived log files, and many other advanced and useful features.

And due to the native *SLF4J* support, we also have the unique option to easily switch to a different logging library at any point if we want to. Overall, the maturity and impressive flexibility of Logback have made it the go-to option next to [Log4J2](#) for most of the Java ecosystem today.

Take a look at some of the most common and detrimental practices that you might encounter when making use of [logging in a Java application](#).



©2018 Stackify
8900 State Line Rd. STE 100
Leawood, KS 66206

At Stackify, we believe that the ideal development team, today and in the future, is consistently optimizing its output across the entire lifecycle of an application; from development to testing to production. With Stackify's [Retrace](#) and [Prefix](#), we hope to support that endeavor.

Along with tools to help improve application performance, we created our Developer Things series to deliver relevant, useful content to developers like you no matter where you are. Check our newsletter before your stand-up, listen to our podcast on your drive home, or read our in-depth explorations of programming when you're ready to improve, expand or validate your skills.

FOR EVEN MORE DEVELOPER
CONTENT,VISIT OUR BLOG AT
[WWW.STACKIFY.COM/BLOG.](http://WWW.STACKIFY.COM/BLOG)