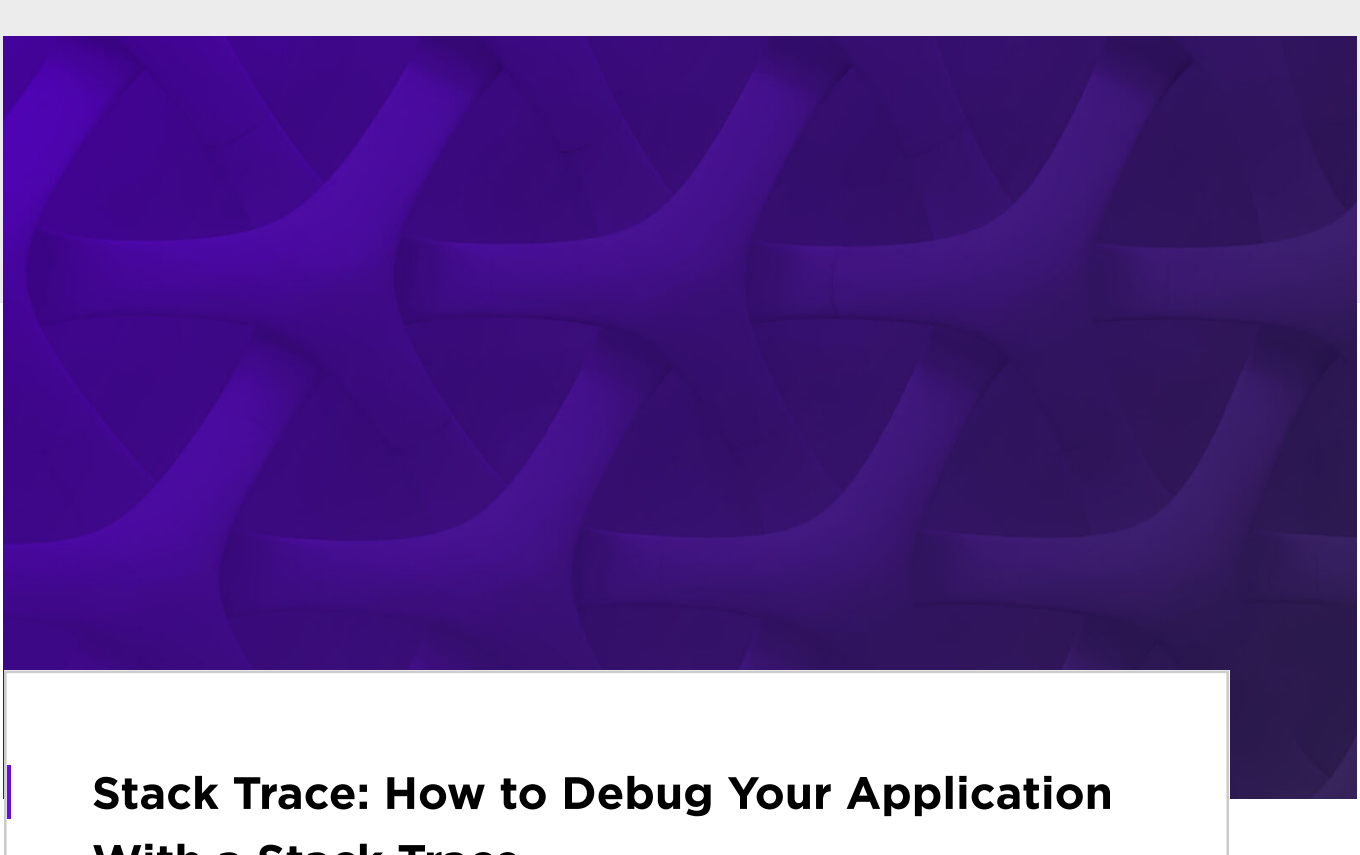


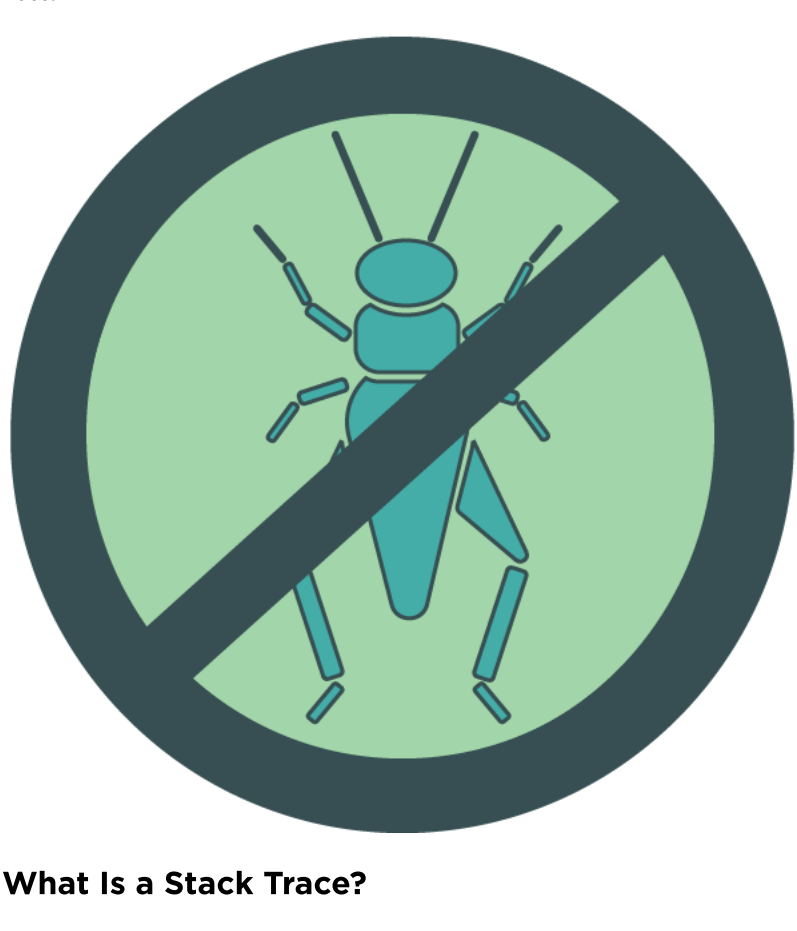
Stack Trace: How to Debug Your Application With a Stack Trace

March 24, 2021
by SentinelOne

As a developer, stack traces are one of the most common error types you'll run into. Every developer makes mistakes, including you. When you mak mistake, your code will likely exit and print a weird-looking message called a **stack trace**.

But actually, a stack trace represents a path to a treasure, like a pirate map. It shows you the exact route your code traversed leading up to the poir where your program printed an exception.

But, how do you read a stack trace? How does a stack trace help with troubleshooting your code? Let's start with a comprehensive definition of a st trace.



What Is a Stack Trace?

To put it simply, a stack trace represents a call stack at a certain point in time. To better understand what a call stack is, let's dive a bit deeper into h programming languages work.

A stack is actually a data type that contains a collection of elements. The collection works as a **last-in, first-out** (LIFO) collection. Each element in t collection represents a function call in your code that contains logic.

Whenever a certain function call throws an error, you'll have a collection of function calls that lead up to the call that caused the particular problem is due to the LIFO behavior of the collection that keeps track of underlying, previous function calls.

This also implies that a stack trace is printed top-down. The stack trace first prints the function call that caused the error and then prints the previo underlying calls that led up to the faulty call. Therefore, reading the first line of the stack trace shows you the exact function call that threw an error

Now that you have a deeper understanding of how a stack trace works, let's learn to read one.

How to Read a Stack Trace

Stack traces are constructed in a very similar way in most languages: they follow the LIFO stack approach. Let's take a look at the Java stack trace below.

```
Exception in thread "main" java.lang.NullPointerException
        at com.example.myproject.Book.getTitle(Book.java:16)
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
```

The first line tells us the exact error that caused the program to print a stack trace. We see a **NullPointerException**, which is a common mistake for programmers. From the Java documentation, we note that a NullPointerException is thrown when an application attempts to use "null" in a case w an object is required.

Now we know the exact error that caused the program to exit. Next, let's read the previous call stack. You see three lines that start with the word "a All three of those lines are part of the call stack.

The first line represents the function call where the error occurred. As we can see, the **getTitle** function of the Book class is the last executed call. Furthermore, the error occurred at line 16 in the Book class file.

The other calls represent previous calls that lead up to the **getTitle** function call. In other words, the code produced the following execution path:

1. Start in *main()* function.
2. Call *getBookTitles()* function in Author class at line 25.
3. Call *getTitle()* function in Book class at line 16.

In some cases, you'll experience chained exceptions in your stack trace. As you could have guessed, the stack trace shows you multiple exceptions have occurred. Each individual exception is marked by the words "Caused by".

The lowest "Caused by" statement is often the root cause, so that's where you should look at first to understand the problem. Here's an example st trace that includes several "Caused by" clauses. As you can see, the root issue in this example is the database call `dbCall`.

```
Exception in thread "main" java.lang.NullPointerException at com.example.myproject.Book.getTitle(Book.java:1
        at com.example.myproject.Author.getBookTitles(Author.java:25)
        at com.example.myproject.Bootstrap.main(Bootstrap.java:14)
Caused by: com.example.ServiceException: Service level error
        at com.example.myproject.Service.serviceMethod(Service.java:15)
        ... 1 more
Caused by: com.example.DatabaseException: Database level error
        at com.example.Database.dbCall(Database.java:39)
        at com.example.myproject.Service.serviceMethod(Service.java:13)
        ... 2 more
```

Hopefully, this information gives you a better understanding of how to approach a call stack. So, what's a stack trace used for?

How to Use a Stack Trace

A stack trace is a valuable piece of information that can be used for debugging purposes. Whenever you encounter an error in your application, you to be able to quickly debug the problem. Initially, developers should look in the application logs for the stack trace, because often, the stack trace t you the exact line or function call that caused a problem.

It's a very valuable piece of information for quickly resolving bugs because it points to the exact location where things went wrong.

Stack trace tracks important metrics that monitor the health of your application.

In addition to telling you the exact line or function that caused a problem, a stack trace also tracks important metrics that monitor the health of you application. For example, if the average number of stack traces found in your logs increases, you know a bug has likely occurred. Furthermore, a lov level of stack trace exceptions indicates that your application is in good health.

In short, stack traces and the type of errors they log can reveal various metrics related to your application as explained in the example.

Common Problems With Stack Traces and Third-Party Packages

Often, your projects will use many third-party code packages or libraries. This is a common approach for languages such as PHP or JavaScript. Ther rich ecosystem of packages maintained by the Open Source community.

In some cases, an error occurs when you send incorrect input to one of the third-party libraries you use. As you might expect, your program will prir stack trace of the function calls leading up to the problem.

However, now that you're dealing with a third-party library, you'll have to read many function calls before you recognize one associated with your cc In some cases, like when too many function calls happen within a third-party package, you may not even see any references to your code.

You can still try to debug your code by looking at where a particular package was used. However, if you use this particular package frequently throughout your code, debugging your application won't be easy.

But there's a solution to the third-party stack problem. Let's check it out!

Solving the Third-Party Stack Trace Problem

Luckily, you can solve third-party stack trace problems by catching exceptions. A call to a third-party library may cause an exception, which will cau your program to print a stack trace containing function calls coming from within the third-party library. However, you can catch the exception in you code with the use of a try-catch statement, which is native to many programming languages such as Java or Node.js.

An effective and simple solution to make your stack traces easy to understand.

If you use a try-catch statement, the stack trace will start from the point where you included the try-catch logic. This presents you with a more actionable and readable stack trace that doesn't traverse into third-party library's function calls. It's an effective and simple solution to make your s traces easy to understand.

On top of that, when your program throws a Throwable instance, you can call the `getStackTrace()` function on the instance to access the stack tra You can then decide to print the stack trace to your terminal so you can collect the information for log analysis. Here's a small example where we th an exception and manually print the stack trace.

```
import java.io.*;

class Sum {

    // Main Method
    public static void main(String[] args) throws Exception {

        try {
            // add positive numbers
            addPositiveNumbers(5, -5);
        } catch (Throwable e) {
            StackTraceElement[] stktrace = e.getStackTrace();

            // print element of stktrace
            for (int i = 0; i < stktrace.length; i++) {
                System.out.println("Index " + i
                                   + " of stack trace contains = "
                                   + stktrace[i].toString());
            }
        }
    }

    // method which adds two positive number
    public static void addPositiveNumbers(int a, int b) throws Exception {
        if (a < 0 || b < 0) {
            throw new Exception("Numbers are not Positive");
        } else {
            System.out.println(a + b);
        }
    }
}
```

The printed output will look like this.

Index 0 of stack trace contains = Sum.addPositiveNumbers(File.java:26) Index 1 of stack trace contains = Sum.main(File.java:6)

Next, let's learn how log management and stack traces work together.

Log Management and Stack Traces

You might wonder what log management and stack traces have to do with each other, and actually, they're very compatible.

It's best practice for your DevOps team to implement a logging solution. Without an active logging solution, it's much harder to read and search for traces. A logging solution provides you with an easy-to-use interface and better filtering capabilities.

A log management solution helps aggregate logs, index them, and make them searchable, all from a single interface. You can run advanced queries find specific logs or stack trace information. This approach is much faster than using the CTRL+F key combination to look through your logs.

Conclusion

To summarize, we focused on the need for a logging solution to access stack traces and any other relevant information your application outputs. A s trace is one of the most valuable pieces of information to help developers identify problems quickly.

Furthermore, a stack trace shows an exact execution path, providing context to developers trying to solve bugs. The first line in the call stack repres the last executed function call, so remember to always read a stack trace top-down. That first function call is responsible for throwing an exception

Want to decrease your bug resolution time? Check out Scalyr's log management solution. If you want to learn more about log formatting and best practices for logging, check out this log formatting article.

Like this article? Follow us on LinkedIn, Twitter, YouTube or Facebook to see the content we post.

Read more about Cyber Security

- Columnar Database: What Is It?