Dashbird.io ++++++++

# 10 THINGS YOU MUST KNOW ABOUT SERVERLESS DEBUGGING

Intro

# WHY ON EARTH SERVERLESS DEBUGGING IS DIFFERENT

Serverless architecture fundamentally changes many aspects of the software development and maintenance life cycle. It's no different when it comes to failure detection and debugging.

Some aspects are uniquely particular to serverless, such as cold-starts and retries, the relationship between function, invocations and logs, etc. We can't properly maintain and debug such applications with the same tools and paradigms we were used to in software running on traditional infrastructures.

Dashbird team has accumulated years of experience in running serverless applications at scale, as well as pitfalls and best practices when it comes to ensuring good health and performance of such systems. Our goal with this e-book was to consolidate this experience and share with the community some of the concepts and practices we consider positive for serverless development teams.

# Summary

# *Application Failures*

We divide failures in a serverless system into two types: runtime and application. We'll get into the first at the next topic. Application failures encompass everything from a syntax error (in case of a scripting language such as Python or NodeJS) to all kinds of execution exceptions.

These could be - generally speaking - a Type error, for example, when a statement is assigning a value of an invalid type to a given variable (e.g. assigning an integer to a string variable); or an Index error, when trying to access the fourth item of a three-element sized list/array, for instance. The list could be very lengthy and will have differences from one programming language to the other. In short, we're talking about everything listed in a programming language's reference docs as an error/exception.

When these errors are raised during the execution of your serverless function, they are considered to be an "application failure". Thorough documentation and official references on the subject will be available for most modern languages, such as Python and JavaScript.

Most cloud providers will offer a logging service attached to their serverless compute offering. AWS Lambda, for example, offers the CloudWatch Logs solution. Although such services will capture and display all your logs, they usually have no embedded intelligence to automatically tell apart what's an application error.

Specialized services - such as Dashbird - will scan your logs and automatically identify application errors based on your programming language. This is convenient because the service can alert you only in case an error is found, saving time when it comes to failure detection by removing the clutter created by other verbose log lines.

# *Runtime Failures*

A runtime failure would be an error associated with the infrastructure or the underlying runtime (the interpreter engine, in case of Python, for example). This will vary according to the serverless platform used and the programming language employed, but here is an illustrative list for AWS Lambda:

## Timeout

Serverless platforms will halt the execution of your function in case it takes too long; as of May 2019, AWS Lambda for example would allow functions to run for up to 15 minutes at maximum.

## Memory Exhaustion

If your function consumes all memory available and demands more, the execution will fail due to the lack of RAM resources to supply your app with.

## Early Exit

This could happen due to an application error, but is a grey area where many crazy things could happen; if you have code compiled to a specific hardware specification, for example, and the machine running your function doesn't meet the requirements, the whole system might crash and exit before your code has a chance to finish or even catch an exception; this could be one of those hard to debug issues.

## Configuration Error

This type of error can be raised when your function is misconfigured some way; it could be that the function needs permission to access an external resource in your cloud stack, but the identify profile attached to it doesn't contain appropriate authorizations, for example.

# *Logging Like a Pro*

When asked “what should I log for debugging purposes?”, the obvious answer would be the exception and the code trace that lead to the error so that we can track down and fix it.

But in case of a serverless application, there are a few other important aspects. The event payload (inputs received by the function) can be paramount when debugging an issue. You would want, for example, to replicate the same request when trying to understand what went wrong, or you might need to understand in which scenarios the issue is manifesting. This would only be possible if you know what was the payload received with the function invocation. For that reason, we strongly advise logging the event payload as default for all executions of all your functions.

Executions associated with specific users/accounts will also benefit from logging whose account it is. It will be important to narrow down which user behaviors are driving issues in your application, for example.

Finally, based on the OWASP Logging Cheat Sheet recommendations, we should be logging: When, Where, Who and What in every function invocation. Which data points would constitute each of these elements will vary according to the application and context, though.

# *Searching for logs*

No matter which serverless platform and logging service you are using, always make sure you have full text and advanced searching capabilities in place. You will need them, at some point, if you want to keep your sanity while debugging issues in your application.

By advanced searching capabilities, we mean tailored to serverless platforms. For example, a microservices approach is usually adopted in conjunction with serverless architectures. In this case, a common scenario is having multiple functions working together to perform a higher level task. When something goes wrong in this task process, there's a good chance you and your team will need to debug two or more functions at the same time. Querying each function's logs history separately can be time-consuming. Look for a service that allows you to search logs across some or all of your functions at once.

# *Execution Logs Individualization*

Serverless platforms usually spin up a new container when they need to serve an invocation, and keep this container alive for some time waiting for possible subsequent requests. The problem is that some services will group logs based on the container lifecycle, not per invocation/request.

When a function is invoked very frequently, a container might last for several minutes, if not hours before it gets recycled by the platform, which will lead to accumulating logs from hundreds or maybe thousands of executions, all bundled together. This could make a developer life very hard when trying to identify which specific invocation resulted in a reported issue, leading to wasted time, money and hurting the dev team's morale.

A simple solution is to individualize logs based on invocation, not container lifecycle. Each log entry will group information output during that unique execution only. When a developer needs to, it will be much easier and faster to navigate those logs. Services like Dashbird automatically reorganize logs from cloud providers - in this case, AWS Lambda - individualizing them by invocation.

# *Retry Behavior*

Some serverless services, such as AWS Lambda, will provide a retry mechanism out of the box. When an invocation results in failure, the platform will automatically retry it with the same parameters. This could repeat multiple times until the operation succeeds or the platform gives up trying. In general, this is a very handful feature, but in some cases can result in unwanted side effects that are not a result of an application or runtime error, but the lack of consciousness of this issue when architecting the application logic.

A customer's credit card could be unwantedly charged twice, for example, because something went wrong in the first execution of a purchase order processing, which got automatically retried by your serverless platform. See the next two sections of the ebook to understand how to make your code resilient to the retry behavior.

In terms of logging, it can benefit your debugging process when you have a service that automatically identifies and connects all the retried executions to the original request, so you can debug them together. Dashbird, for example, automatically does that out of the box for all your Lambda functions.

# *Idempotency*

Idempotency is a “property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application”1.

Let’s say we have a function like this:

```
def multiply_by_two(number):
    x = number * 2
    return x
```

This function is an example of an idempotent one. No matter how many times you execute it with the same “number” argument, it will always produce the same result and no undesired side effects.

Now consider a function that receives a list of items from an e-commerce shopping cart and creates an order for a customer. If not implemented properly, executing this function multiple times will end up creating multiple orders, although that might not be what we would want to do.

Read operations usually do not produce any side effects, they’re idempotent by nature. Storing and deleting a value aren’t idempotent operations by nature, but they can be if we have a unique identifier (UID) for that resource. In our e-commerce scenario, if the customer order has a UID, the storing operation can be performed multiple times without creating multiple different order placements.

The order UID could be, for instance, a hash of the customer email or username, the purchase timestamp, and a list of items purchased. These variables would be sent as a parameter to our API when the site receives the order request. If the function fails at some point and the invocation is retried, the same order UID would be generated again, meeting the idempotency requirement. This is just for illustration purposes, each circumstance will require proper analysis to find a stable and resilient idempotent implementation.

1 https://en.wikipedia.org/wiki/Idempotence

# *Dead Letter Queue*

Serverless platforms that provide the retry feature will auto-execute your function in case of failure, but only up to a few times, otherwise, they could enter an infinite loop. When the last try still results in a failure, usually they will provide a configuration option for what to do with that request. It could just be forgotten, which might not be clever in many cases, or you could send those failed requests to what's called a dead letter queue (DLQ).

Once in the DLQ, they would be stored for some time, maybe days or weeks, giving you the opportunity to investigate the issue, fix it and then run the requests again. Make sure you analyze your application requirements and setup proper DLQ configuration wherever that makes sense.

# *Separation of Concerns*

Usually a good practice in any software project, but especially important in serverless apps, is the separation of concerns2. This is usually applied to the codebase: making sure that each class or function, for example, is addressing a well defined and single concern at a time. When it comes to serverless, this could be applied to the infrastructure-level by ensuring each function takes care of a single concern.

In an e-commerce site, for instance, Instead of having a single function taking care of:

1. Receiving purchase order request;
2. Making sure items are available in stock;
3. Charging credit card;
4. Dispatching notification to warehouse & packaging operational staff;
5. Sending confirmation email to the customer;

A better implementation would be having a single serverless function taking care of each of those concerns. Maybe each of them could still be split up in smaller services. For example, you could have a general purpose “sendMail” function, which would be called by the “orderConfirmation” function to send a message to a particular person following a pre-defined template. Architectures following this kind of principle would, in general, also be called microservices.

2 https://en.wikipedia.org/wiki/Separation_of_concerns

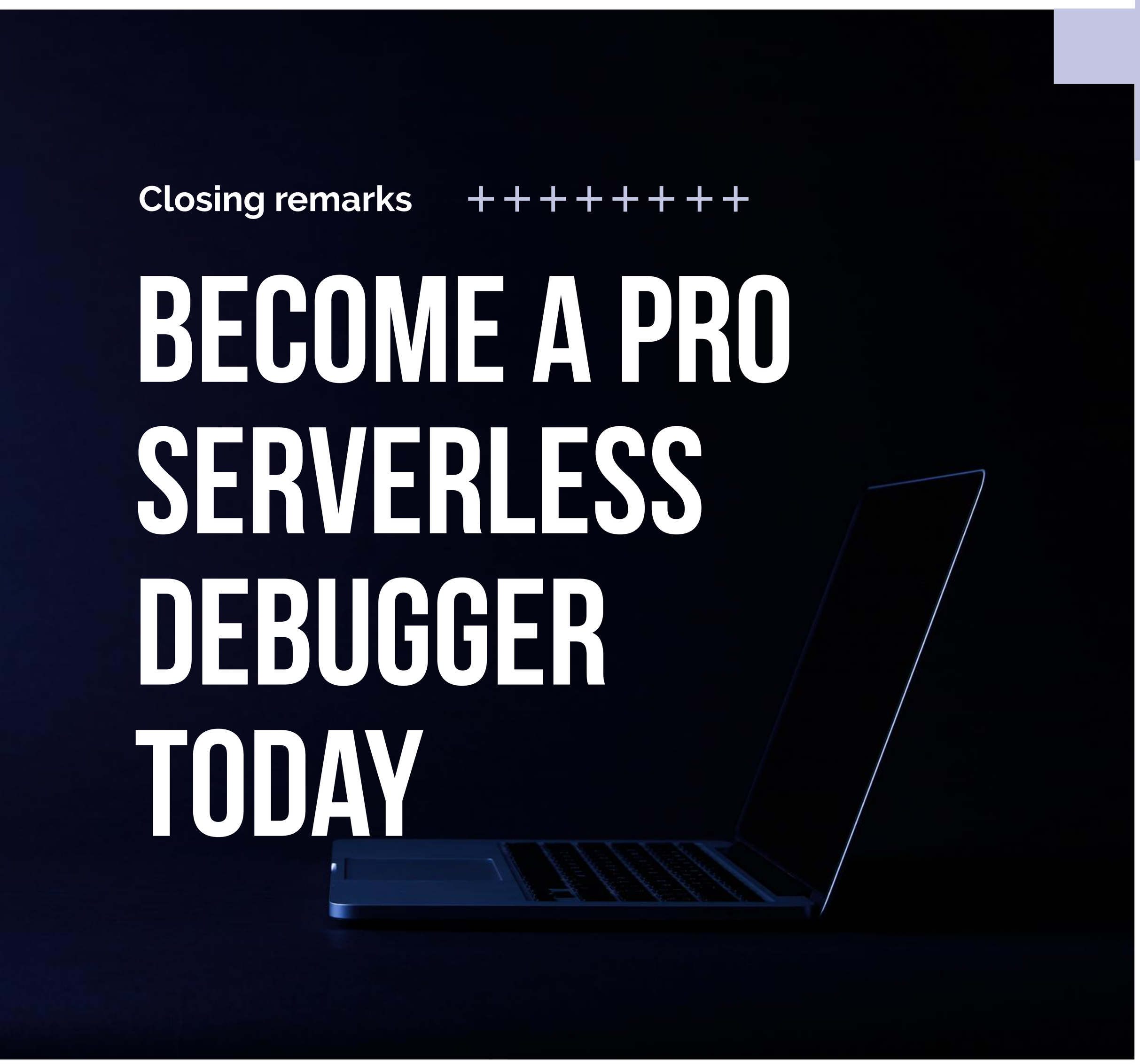
# *Security Logging*

When it comes to securing a serverless application, some good logging practices can be crucial. We already covered the logging of event payload. Another aspect that could come in hand when your application is under attack is logging the response payload. By investigating this data, you will be able to narrow down to the motivation of an attacker, for example. In the sad event of a data breach, it will need to know which data points are now in possession of a malicious actor as well.

Some types of failures can also be very helpful to act proactively. Let's say that usually, 0.5% of the login requests in your application result in a "password don't match" response. If this number goes up to, say, 10% it's definitely a red flag for someone trying to brute force your system and hack your user's accounts.

Dashbird, for example, offers a monitoring aggregation feature that allows receiving proactive alerts whenever a given metric reaches a certain threshold, like in the authentication failure percentage example above. This would fall more in the "monitoring" field, not particularly "logging", so we'll cover more details in another e-book. It is important, though, that a good strategy for monitoring your application is backed by appropriate logs for further debugging and fixing issues or addressing security flaws.

Closing remarks




Dashbird was especially designed from the ground up to provide the best debugging experience for serverless developers using AWS Lambda. If you are looking for a way to professionally monitor your serverless applications in production, quickly detecting and acting upon issues, you should definitely check it out. You can try our full set of features for a trial period or even use our service entirely free, in case your needs aren't that big.

Dashbird: https://dashbird.io