Everybody cooks beans! Get good at debugging!

#webdev #beginners #debugging

TLDR: Inevitably, you will create bugs. But can you debug it effectively? That’s what will matter.

Introduction

Different organizations have their jargon. One I’ve picked up, stemming from my Engineering Manager is **"Everybody cooks beans"**, which aims to communicate that everybody makes mistakes.

This statement hopes to present a mentality that as much as we all want to do good work, mistakes are inevitable. Although his focus is on software engineering, it isn’t exclusive to the field. An adventure into any field presents opportunities to make mistakes. And it's in those mistakes that learning happens.

Software engineering christened the term "bugs" to refer to unintended behaviors from a particular program. "Unintended" is the keyword here, as software essentially acts as designed not as intended.

Code doesn’t lie.

As software engineers, our role is to convert intention into code using syntax from a myriad of languages. This skill of delivering precise instructions to a computer to perform tasks comes with a responsibility to translate intention into instructions. It doesn't take long to realize that most of your work as a software engineer is to make the program behave as it's supposed to rather than as it is😅

Precise instructions create features, the alternative produces bugs.

This isn't always the case as some faults in a program might impact performance but not obstruct function.

NB: I don’t attempt to distinguish between bugs and errors. In my view, the disparity between the two, if any, wouldn’t significantly impact the focus of this article. As such I use the terms **bugs** and **errors** interchangeably.

Some common types of bugs/errors can be found here.

As software engineering grows and as software becomes an interconnected network of services, the surface area for bugs to exist has only widened.

Debugging which is analogous to troubleshooting is the process that involves identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it.

The discipline to traverse multiple factors to narrow down issues to get at a root cause stands as one of the most important skills a developer can have.

The skill to debug software, therefore, isn't an add-on, it's a must-have.

Debugging your own software has been described as being a detective in your own crime.

Even the most experienced and skilled developers are susceptible to committing crimes of their own. The only difference exists in that, through time, they have sharpened their detective skills to more quickly figure out the culprit.

Some of the most advanced organizations still fall prey to bugs leading to an inability to dispense their services. A common term used in situations such as these is "outage".

These can be caused by internal or external factors. The interconnectivity of services testifies to the inevitability of bugs/errors per time. Even the likes of Mozilla, Cloudflare and AWS suffer outages. Usually, these are followed up by a post-mortem which, essentially, is an analysis of the error with the benefit of hindsight.

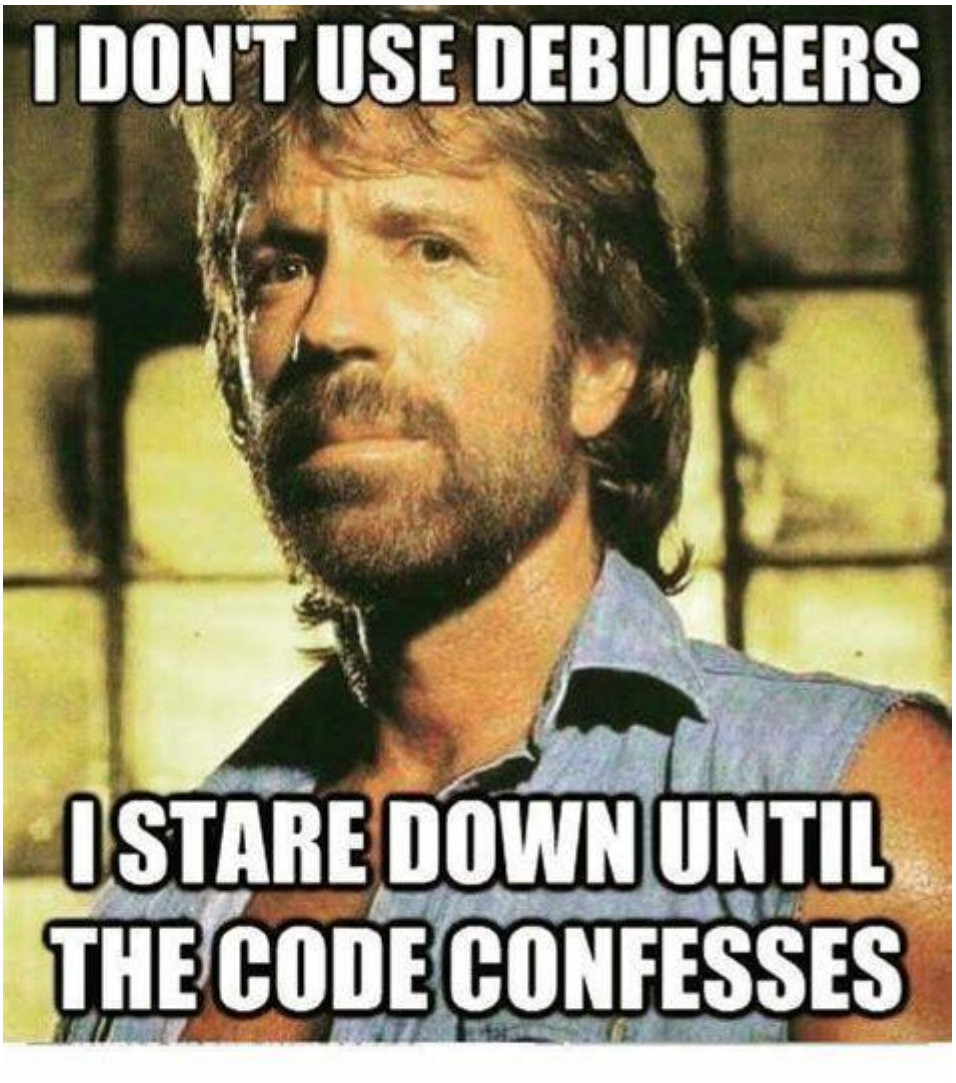
A comprehensive list of post-mortems can be found here.

Challenges of debugging

Compared to other aspects of software engineering there aren't as many well-documented resources on debugging. Worse is that developers often have an informal introduction to debugging, which is in the instance of a fault.

Quite often, debugging happens under pressure and with little information to go on. Failures are often sporadic. Engineers are short on time and have to come up with a solution yesterday. Managers check-in within intervals. In worse cases there's panic, and everyone eagerly waits to hear "we fixed it".

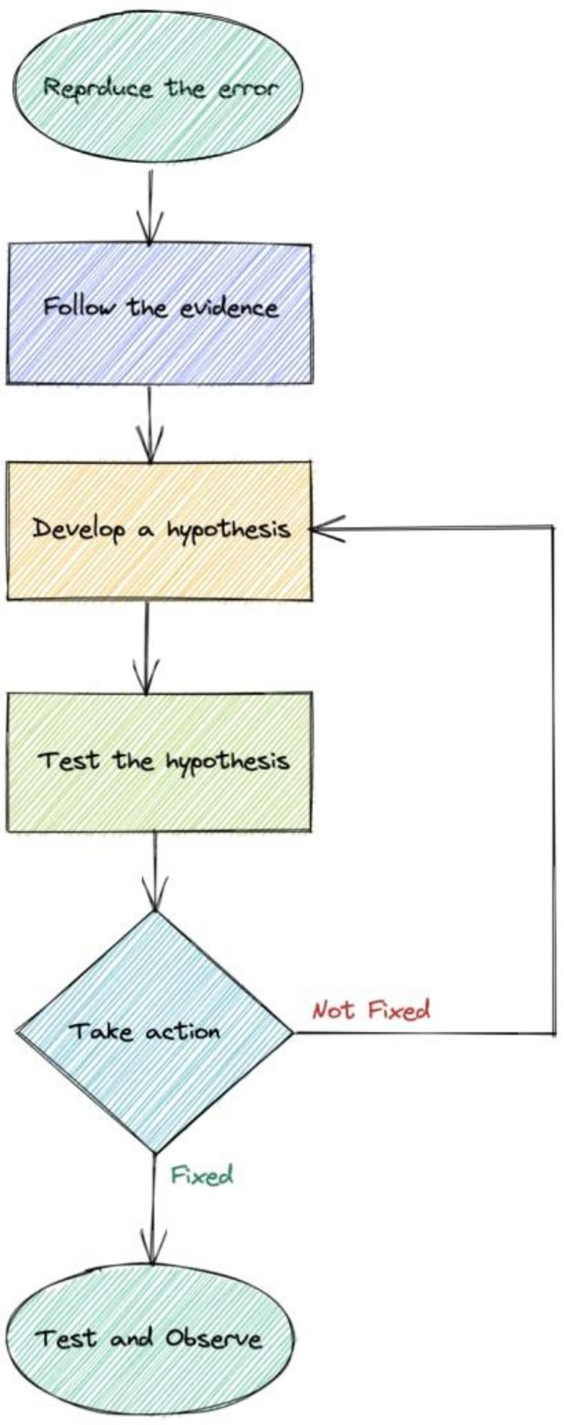
In such cases, it's important for the developer to follow a systematic approach. This entails repeatable steps that follow a predefined pattern. Following a systematic approach will help other developers too, should they encounter a similar bug.



Methodology

In no manner do I consider myself an expert at debugging but I have developed repeatable patterns that have helped me in tracking down and fixing bugs. It’s my opinion on debugging which bears semblance to the scientific method.

Note: The majority of my experience is as a front-end developer in the webspace as such my examples would be given in that regard. However, I do believe that this approach would benefit other aspects of software engineering. More so, I'm aware that there are caveats in peculiar cases for which one or more steps in my method would not apply but **my focus is on the rule and not the exceptions**.



Step 1: Reproduce the error

This is a critical step and **must not** be overlooked. You have to be sure that a particular sequence of steps or a certain scenario causes the bug and document it. You'd need it later.

Go ahead to detail the bug using the following guide:

- Describe the bug clearly

- Give a detailed account of the environment
- Outline precise steps to reproduce the bug
- Include attachments for clarity
- This is called bug reporting and an elaborate guide to writing good bug reports can be found here.

If you lack sufficient information about the bug, try to get some before you forge ahead, otherwise, you'd be flying blind.

A bug that cannot be reproduced cannot be said to have been fixed.

There might be instances where reproducing the error is impossible or painfully difficult. There are measures to put in place to aid debugging in such cases and would be discussed under the developments in debugging section. For now, move on to the next step.

For web developers, a platform like Code Sandbox is invaluable in debugging as it provides an isolated environment where the error can be reproduced reliably. No longer would we be haunted by statements like "it fails on my computer".

Step 2: Follow the evidence

Analyze the call stack and error message. Different programming languages provide a means to access the call stack in the error object. This provides a stack trace that contains information related to the failure.

Usually, the stack trace contains the following information:

- file name where the error occurred
- line number in the file where the error occurred
- the column number in the file where the error occurred
- function/class method name in which error occurred

Going through the logs is vital as stack traces can provide insight into the origin and type of the error.

It'd also be beneficial to know how to use Google. Query structure and google docs help a lot in narrowing down searches. You can learn that here.

If need be, go through documentation and test files. This should become second nature to software engineers with the inevitability of third-party software. Once, I had to figure out the functionality of a library by going through their tests because the documentation sucked.

In the case of third-party software, there'd have been other developers who've encountered this error. If so, you can short-circuit the process and skip to step 4.

However, if you're the unlucky fellow 😅 that encounters the issue first proceed to step 3.

Step 3: Develop a hypothesis

At this juncture, based on the knowledge of the systems at play as well as their interconnections, you should make an educated guess.

Mark Erikson, a Redux maintainer, outlines this as having a plan,

Understand how the code should behave first. Look at the actual behavior. You can probably make educated guesses for where things are diverging. Focus on those areas. Don't randomly tweak. Change one thing at a time and compare results.

Experience definitely helps here, but even if you're new to the situation there's no need to fear. Come up with an explanation according to what you currently understand. You can always revise it later.

This hypothesis can come from colleagues as well as from Q&A forums like stack overflow as well as search results from Google or tutorials from Youtube.

Step 4: Test the hypothesis

It is possible, based on prior knowledge, to jump right through by guessing the cause of failure. However, I have seen that this isn't always possible. As such any proposed solution should be applied and then validated against the steps in the bug report.

Do not be a sucker for punishment! First, try the suggestions from your search results and Stack Overflow. Especially for third-party software.

Step 5: Take action

Treat the disease and not the symptoms.

Here you can choose to refined the method of resolving the bug. This can be to refine your hypothesis based on info gotten from testing the hypothesis.

Check if the bug is a stand-alone issue or if it's a result of underlying problems in the codebase or configuration. If you're satisfied with the outcome of testing the hypothesis then you can commit to the fix (for example to make a pull request).

Step 6: Test and Observe

"Never allow the same bug to bite you twice." - Steve Maguire

There's the need to be watchful as a supposed fix might introduce new bugs. Therefore, regressive testing should be carried out to determine if the bug really had been squashed.

Regression testing is a software testing practice that ensures an application still functions as expected after any code changes, updates, or improvements.

Herein lies the importance of Step 1: Reproducing the bug. You'd have no way of knowing if truly the bug has been fixed if you didn't have procedures that cause the bug to surface.

Test with those procedures and others to confirm, and if you discover another bug, Great! It means you caught a bug before it went into production.

Testing isn't debugging. Testing here starts with a known condition (testing whether the image shows up) while debugging starts with an unknown condition (why is the image not showing up). Another difference, although somewhat comical, is that managers would accept a test that yields no fault but not a debugging effort that yielded no solution 😅

A case study

Let's consider a scenario where this methodology was followed.

NB: My example might seem trivial but the aim was to details a scenario where the methodology could be seen in action. Also to point out that bugs/errors might surface from little things.

I recently encountered a bug/error, where the Twitter preview image wasn't showing up for a specific page of a website.

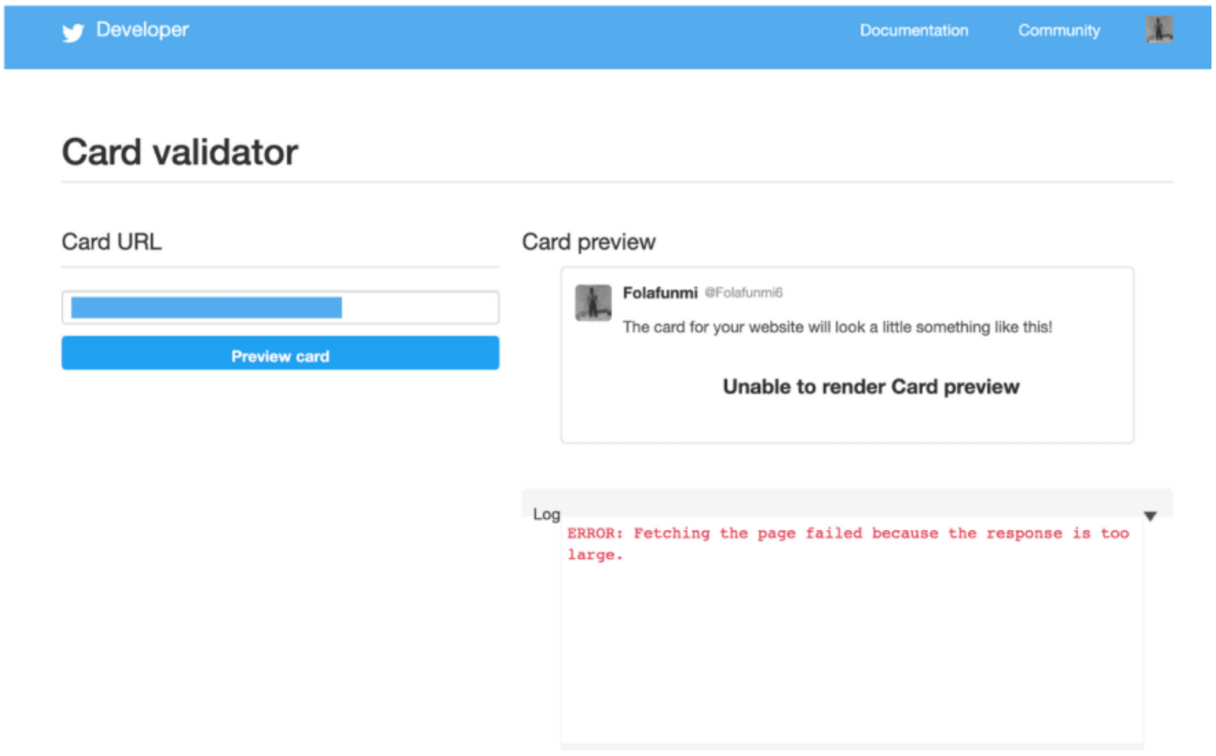
Step 1: Reproduce the error

The bug report can be outlined as:

- **Description:** Twitter preview image not showing up for landing page route alone.
- **Environment:** Twitter web and mobile
- **Steps to reproduce:** Tweet *** link, the preview doesn't show up.
- **Attachments:** (excluded for this example)

Step 2: Follow the evidence

Firstly, I tried to validate the card preview and got an error.



```
ERROR: Fetching the page failed because the response is too
large.
```

ERROR: Fetching the page failed because the response is too large.

Step 3: Develop a hypothesis

On seeing the words "response was too large", I checked to confirm the resolution of the image was indeed hosted on the site, it was about 1200x1200. Hmmm…seems large🤔

Hypothesis: The image is too large and twitter's bot isn't able to fetch it.

Step 4: Test the hypothesis

I requested for an image with smaller dimension and waited.

While waiting I happened to run across the description and noted that only one page had this issue and the same image had been used for the others. I checked the docs for Twitter's limit on the dimension of a preview image, it was 4096x4096.

What? So the image fits within the limit?! 🤦🏾

I didn't bother checking if it'd be fixed in Step 5, it wouldn't.

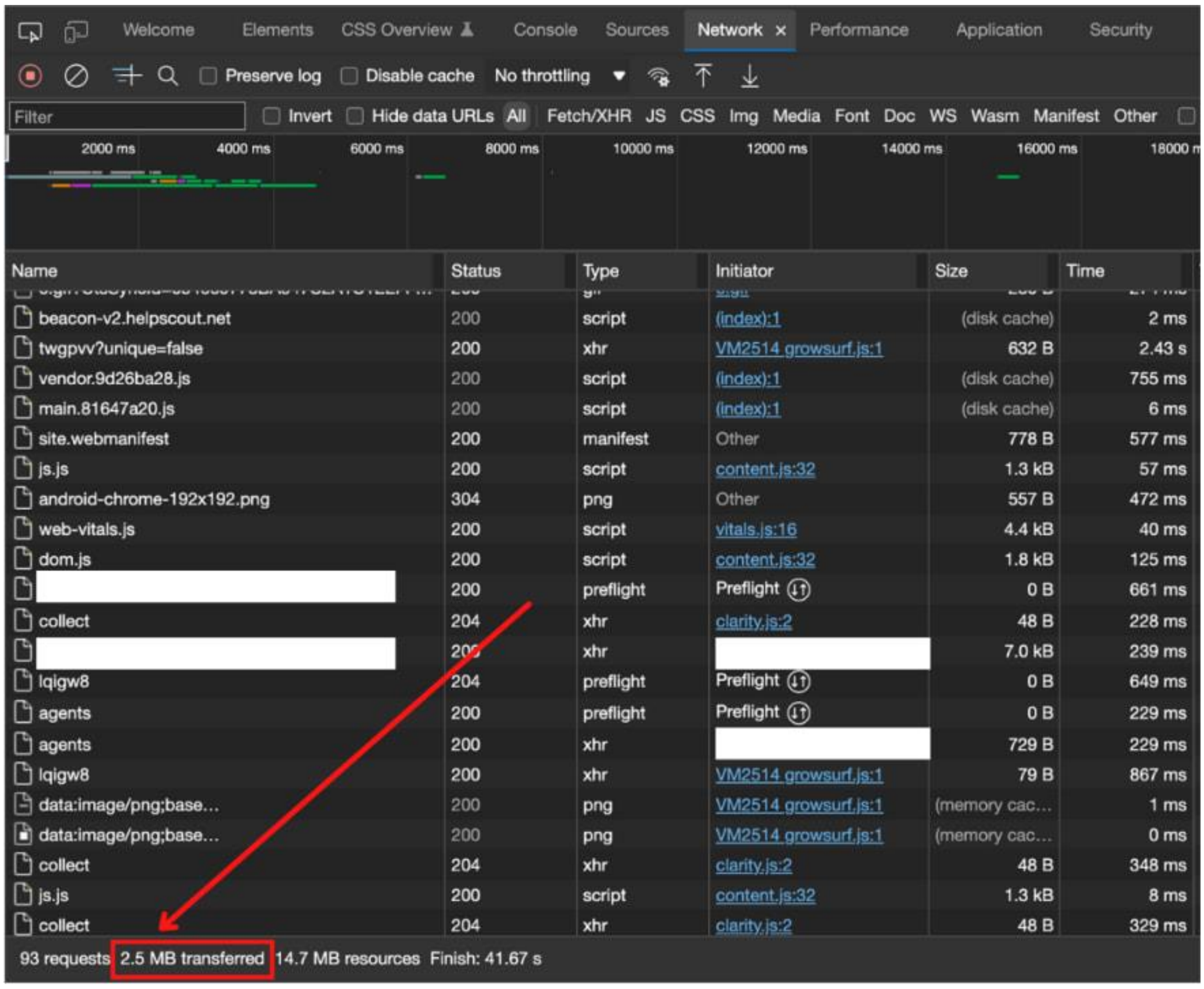
Back to Step 3: Develop a hypothesis

I read the error message again.

ERROR: Fetching the page failed because the response is too large.

Oh, the *page* itself is too large?

I checked the documentation and found that the limit for page size is 2MB. I thought that was ridiculous. No way was the page document over 2MB. However, I check the network tab for the response.

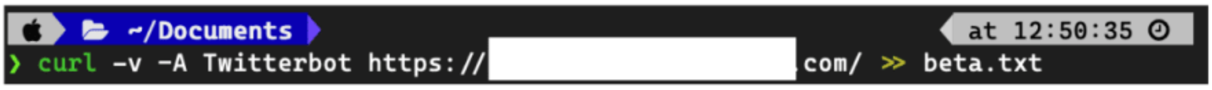


Name	Status	Type	Initiator	Size	Time
beacon-v2.helpscout.net	200	script	(index):1	(disk cache)	2 ms
twgpvv?unique=false	200	xhr	VM2514 growsurf.js:1	632 B	2.43 s
vendor.9d26ba28.js	200	script	(index):1	(disk cache)	755 ms
main.81647a20.js	200	script	(index):1	(disk cache)	6 ms
site.webmanifest	200	manifest	Other	778 B	577 ms
js.js	200	script	content.js:32	1.3 kB	57 ms
android-chrome-192x192.png	304	png	Other	557 B	472 ms
web-vitals.js	200	script	vitals.js:16	4.4 kB	40 ms
dom.js	200	script	content.js:32	1.8 kB	125 ms
[redacted]	200	preflight	Preflight	0 B	661 ms
collect	204	xhr	clarity.js:2	48 B	228 ms
[redacted]	200	xhr	[redacted]	7.0 kB	239 ms
lqigw8	204	preflight	Preflight	0 B	649 ms
agents	200	preflight	Preflight	0 B	229 ms
agents	200	xhr	[redacted]	729 B	229 ms
lqigw8	200	xhr	VM2514 growsurf.js:1	79 B	867 ms
data:image/png;base...	200	png	VM2514 growsurf.js:1	(memory cac...	1 ms
data:image/png;base...	200	png	VM2514 growsurf.js:1	(memory cac...	0 ms
collect	204	xhr	clarity.js:2	48 B	348 ms
js.js	200	script	content.js:32	1.3 kB	8 ms
collect	204	xhr	clarity.js:2	48 B	329 ms

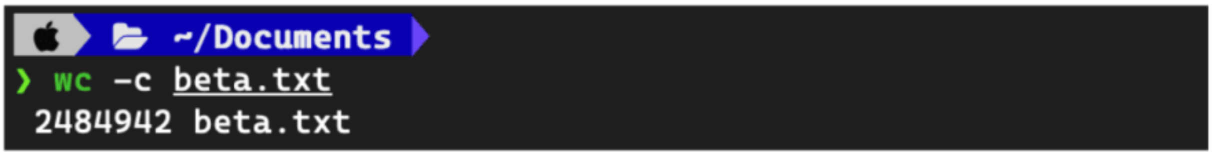
93 requests | 2.5 MB transferred | 14.7 MB resources | Finish: 41.67 s

What?! 2.5MB?! How's that possible, it's a document?!

The site was built using React so I couldn't check the file size directly. I had to see what the Twitter bot saw, so I went to the command line.



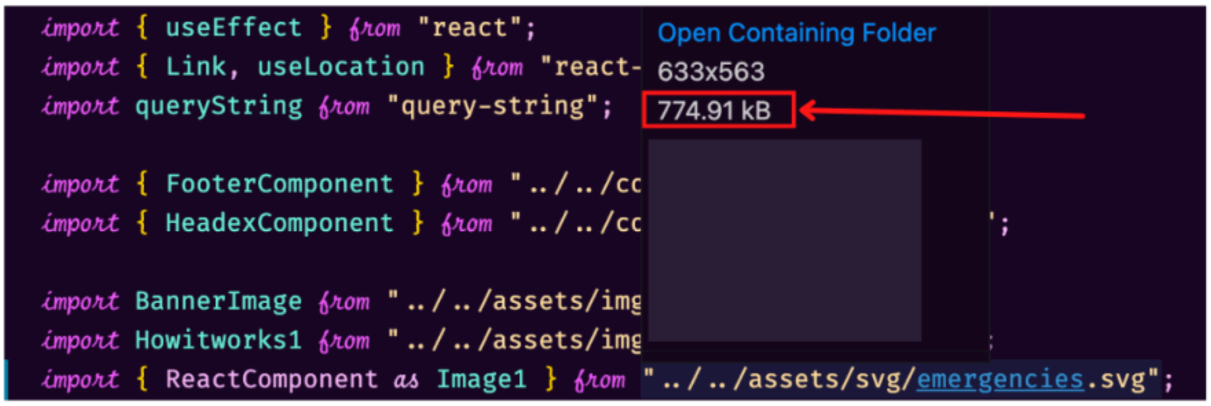
```
 ~/Documents                                              at 12:50:35
❯ curl -v -A Twitterbot https://[redacted]com/ >> beta.txt
```



```
 ~/Documents
❯ wc -c beta.txt
 2484942 beta.txt
```

Wow, so the document really was 2.5MB. How so?

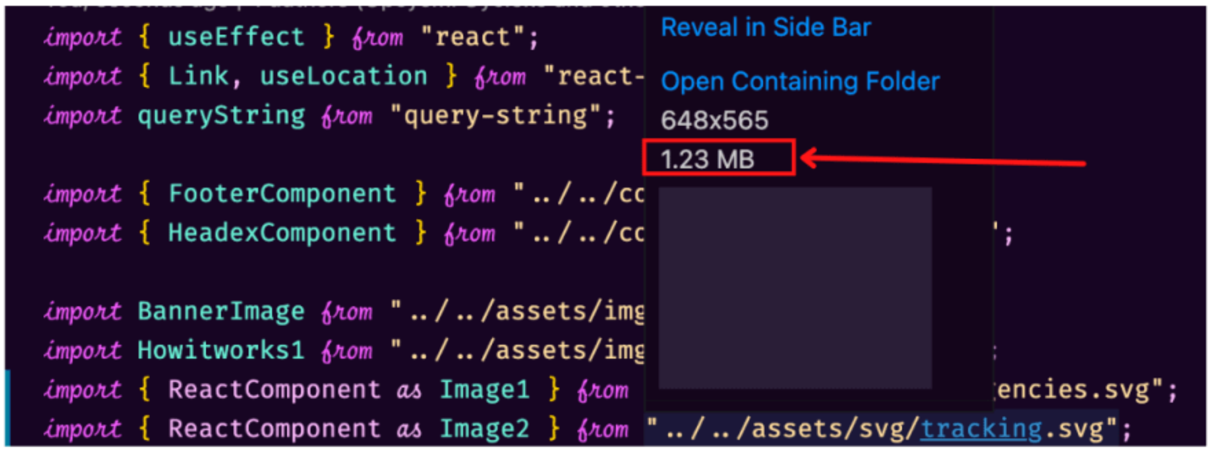
I went snooping around the page source (pretty printed of course) but I wasn't sure what I was looking for. So I went to VScode and opened up the component for that page. Going through the imports I found something interesting.



```
import { useEffect } from "react";
import { Link, useLocation } from "react-
import queryString from "query-string";

import { FooterComponent } from "../../cc
import { HeadexComponent } from "../../cc                                ';

import BannerImage from "../../assets/img
import Howitworks1 from "../../assets/img                                ;
import { ReactComponent as Image1 } from "../../assets/svg/emergencies.svg";
```



```
import { useEffect } from "react";
import { Link, useLocation } from "react-
import queryString from "query-string";

import { FooterComponent } from "../../cc
import { HeadexComponent } from "../../cc                                ';

import BannerImage from "../../assets/img
import Howitworks1 from "../../assets/img                                ;
import { ReactComponent as Image1 } from                            encies.svg";
import { ReactComponent as Image2 } from "../../assets/svg/tracking.svg";
```

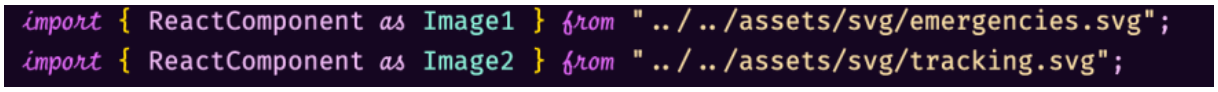
The SVGs were large (not good), but that wasn't what got to me. They were imported as `ReactComponent`. This meant the image would actually be rendered as svg elements. And being that size the elements would be massive.

Aha!! I found the culprit!!

Step 4: Test the hypothesis

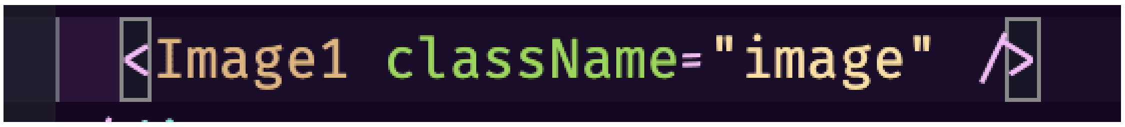
I then switched out the imports and changed the way the images were rendered from using the `svg` tag to the `img` tag.

That's from the old import,



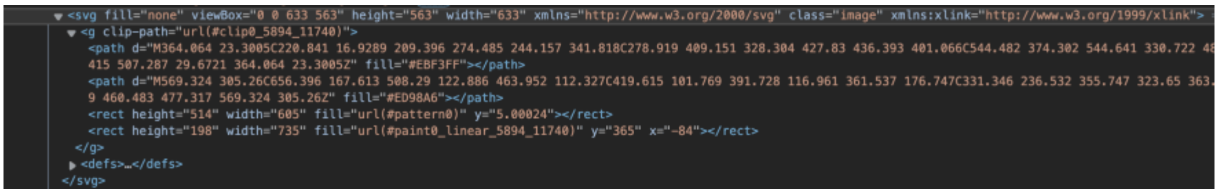
```
import { ReactComponent as Image1 } from "../../assets/svg/emergencies.svg";
import { ReactComponent as Image2 } from "../../assets/svg/tracking.svg";
```

And the old usage,



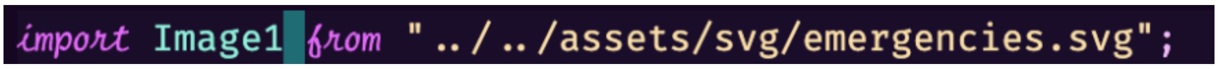
```
<Image1 className="image" />
```

Which is then rendered as,



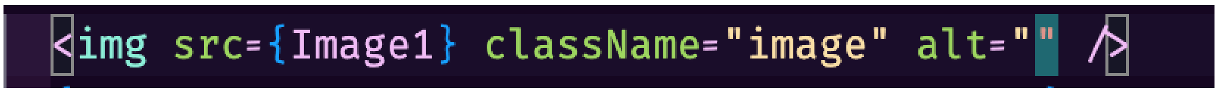
```
▼ <svg fill="none" viewBox="0 0 633 563" height="563" width="633" xmlns="http://www.w3.org/2000/svg" class="image" xmlns:xlink="http://www.w3.org/1999/xlink">
  ▼ <g clip-path="url(#clip0_5894_11740)">
      <path d="M364.064 23.3005C220.841 16.9289 209.396 274.485 244.157 341.818C278.919 409.151 328.304 427.83 436.393 401.066C544.482 374.302 544.641 330.722 48
      415 507.287 29.6721 364.064 23.3005Z" fill="#EBF3FF"></path>
      <path d="M569.324 305.26C656.396 167.613 508.29 122.886 463.952 112.327C419.615 101.769 391.728 116.961 361.537 176.747C331.346 236.532 355.747 323.65 363.
      9 460.483 477.317 569.324 305.26Z" fill="#ED98A6"></path>
      <rect height="514" width="605" fill="url(#pattern0)" y="5.00024"></rect>
      <rect height="198" width="735" fill="url(#paint0_linear_5894_11740)" y="365" x="-84"></rect>
    </g>
  ▶ <defs>…</defs>
  </svg>
```

To the new import,



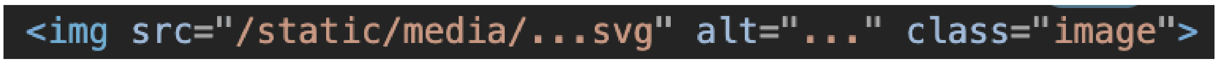
```
import Image1 from "../../assets/svg/emergencies.svg";
```

And the new usage,



```
<img src={Image1} className="image" alt="" />
```

Which is then rendered as,



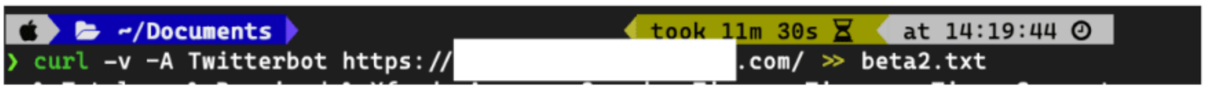
```
<img src="/static/media/...svg" alt="..." class="image">
```

A simple change but this ensured the `svg` tag stayed out of the picture.

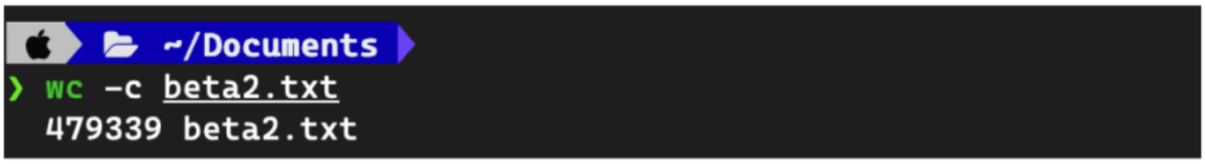
I could also have requested for smaller SVGs too.

Step 5: Take action

I made the PR and after some explanation it got merged into beta. I went back to the command line,



```
 ~/Documents                                  took 11m 30s   at 14:19:44
❯ curl -v -A Twitterbot https://[redacted].com/ >> beta2.txt
```



```
 ~/Documents
❯ wc -c beta2.txt
  479339 beta2.txt
```

Hurray!!! 479kB!!!

Step 6: Test and observe

Went back to the Twitter card validator and confirmed that the preview image showed up. It did!!

Tests

Different types of tests include:

- **Unit:** Verify that individual, isolated parts work as expected.
- **Integration:** Verify that several units work together in harmony.
- **End-to-End(e2e):** A helper robot that behaves like a user to click around the app and verify that it functions correctly. Sometimes called "functional testing" or e2e.

We're to write tests that give confidence in our software, and this presents a challenge with testing.

In reference to this, Kent C Dodds expatiates on a tweet on writing tests by Guillermo Rauch.

Guillermo Rauch

@rauchg

Write tests. Not too many. Mostly integration.

16:43 PM - 10 Dec 2016

You can read about that here.

Developments in Debugging

No one likes frustration. As a result there have been advancements in debugging tools to improve the developer experience as touching tracking and fixing bugs/errors.

No one wants to be this guy.

SPENDS 2 WEEKS FIXING A HUGE BUG IN CODE THAT APPEARS TO BE UNFIXABLE
PROBLEM: MISSING PERIOD

As such tooling has been evolved to reduce the surface area for bugs to reside. Some of these developments include:

- Better IDE support for debugging
- Static analysis
- Logging
- Open telemetry
- Command-line tools
- Profilers
- CI

Communities

In all honesty, I have found communities to be the most helpful resource when debugging. It's pacifying to know that someone else ~~shares your pain~~ has encountered a similar problem. It's even better to know that one or more solutions have been implemented.

The feeling of copy-pasting an error message and seeing several matching results from Stack Overflow is just soothing.

Software engineering as a whole has benefited greatly from the sharing of knowledge. Thereby, removing the need to reinvent the wheel.

Conclusion

Most developers would confess that it's a fool's errand to hope to write bug-free code. Rather, it's more practical to write resilient software that can be debugged more easily.

Similarly, good system designs bake in fault tolerance from the get-go. In reality, it's not **if** they will occur rather **when** will they occur. The most experienced engineers also have their demons.

It takes time and practice. You have to get better at reading error messages and *Googling*. But over time it gets easier and is a necessary skill to grow. Don't allow early frustrations to keep you from trying. You'd struggle more at the beginning, don't worry everyone does.

Everyone eventually cooks beans, but can you debug it effectively? That's what will matter.

Hope this article helps.

Resources

1. https://m-cacm.acm.org/magazines/2018/11/232215-modern-debugging/fulltext
2. https://www.geeksforgeeks.org/software-engineering-debugging-approaches/
3. https://kentcdodds.com/blog/write-tests https://kentcdodds.com/blog/static-vs-unit-vs-integration-vs-e2e-tests
4. https://isocroft.medium.com/get-better-at-troubleshooting-and-debugging-software-11d0edd4dac5
5. https://medium.com/mindorks/bug-smashing-a-guide-to-debug-your-app-11278d832e13
6. https://blog.isquaredsoftware.com/2019/01/blogged-answers-debugging-tips/
7. https://flaviocopes.com/debugging/
8. https://www.jotform.com/blog/bug-report/
9. https://siderlabs.com/blog/11-types-of-software-bugs/
10. https://www.freecodecamp.org/news/what-is-a-software-post-mortem/
11. https://www.katalon.com/resources-center/blog/regression-testing/